TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

KAROLIINA JÄSPI

TESTAUSSTRATEGIAN LUOMINEN OLEMASSA OLEVAAN WEB-SOVELLUKSEEN

Diplomityö

Tarkastaja: prof. Tommi Mikkonen
Tarkastaja ja aihe hyväksytty
Tieto- ja sähkötekniikan tiedekunta-neuvoston kokouksessa 14. tammi-kuuta 2015

TIIVISTELMÄ

KAROLIINA JÄSPI: Testausstrategian luominen olemassa olevaan web-sovellukseen
Tampereen teknillinen yliopisto
Diplomityö, 60 sivua
Kesäkuu 2015
Tietotekniikan diplomi-insinöörin tutkinto-ohjelma
Pääaine: Ohjelmistotuotanto
Tarkastaja: professori Tommi Mikkonen

Avainsanat: web-sovellus, web-sovelluksen testaaminen, testaus, testausstrategia

Internetin kehittyessä web-sivustot ovat muuttuneet web-sovelluksiksi, joiden laatuvaatimukset eroavat perinteisistä sovelluksista. Web-sovellusten dynaamisuus asettaa myös uudenlaisia haasteita web-sovellusten testaamiselle. Vanhat testauskäytännöt eivät sovellu hyvin web-sovellusten testaamiseen ja web-sovellusten testausta kehitetäänkin jatkuvasti eteenpäin.

Tässä diplomityössä on luotu testausstrategia kohdeyritykselle. Testausstrategia sisältää suunnitelman siitä, miten yrityksen web-sovellusta testataan toiminnallisen ja ei-toiminnallisen testauksen avulla. Testausstrategiaa suunniteltaessa rajoitteena olivat yrityksen pienet resurssit testausstrategian luomiseen ja varsinaiseen testausprosessiin. Erityisenä haasteena testausstrategian luomisessa oli se, että järjestelmää oli jo kehitetty noin 10 vuotta ja vakiintuneita testauskäytäntöjä oli vähän.

Kohteena olleeseen web-sovellukseen luotiin testausstrategia, joka keskittyi testaamaan kohdesovellusta soveltamalla eri järjestelmätestauksen muotoja. Testausprosessiin suunniteltiin tuotavaksi käyttöliittymän kautta tehtävää testiautomaatiota, jolla saataisiin vähennettyä ennen paljon aikaa vienyttä regressiotestausta. Testiautomaatiota ei päästy kehittämään kunnolla työn aikana, koska järjestelmään tehtiin suuri käyttöliittymämuutos. Versiojulkaisun aikataulun kireyden takia automaatiotestejä ei ehditty päivittämään käyttöliittymämuutoksen valmistumisen jälkeen.

Yrityksen testausprosessia saatiin testausstrategiaa soveltamalla tehostettua. Testausstrategiaa kehitettäessä löydettiin ongelmia useammasta yrityksen prosessista. Vaikka testausstrategiaa ei saatu kehitettyä niin paljon kuin oli suunniteltu, työn lopputuloksena pystyttiin esittelemään yritykselle monia parannuskohteita.

ABSTRACT

KAROLIINA JÄSPI: Creating a testing strategy for an existing web application
Tampere University of Technology
Master of Science Thesis, 60 pages
June 2015
Master's Degree Programme in Information Technology
Major: Software Engineering
Examiner: Professor Tommi Mikkonen

Keywords: web application, web application testing, testing, testing strategy

Due to advancement of the Internet, static web pages have evolved into dynamic web applications. Their quality requirements are different from those of traditional applications and their dynamic nature creates new types of challenges for software testing. Traditional testing conventions are poorly compatible with web application testing and for that reason development of web application testing practices is a constant effort.

This thesis discusses the creation of a testing strategy for a company. The testing strategy contains a plan for functional and non-functional tests for the company's web application. Resources for the creation of the strategy and the actual testing were limited. Additional challenge was that the web application has been developed for approximately 10 years and existing testing conventions were sparse.

The main focus of the created testing strategy was on different forms of system testing. The testing process was originally planned to contain automated testing through the user interface. This would have helped reduce cumbersome and time-consuming manual regression testing. The development of automated tests did not succeed, because the application underwent a significant user interface overhaul. Due to the tight schedule of the version release, there was no time to update the automated tests when the user interface overhaul was completed.

Overall, the testing strategy helped to enhance the testing process of the target company. In the development of the testing strategy, problems were uncovered in many of the company's processes. Even though the development of the testing strategy was not as complete as planned, many areas of improvement were discovered and presented to the company as a result of the thesis.

ALKUSANAT

Esitän kiitokset yritykselle, joka tarjosi mahdollisuuden tehdä heille diplomityön. Erityiset kiitokset haluan antaa työni ammattitaitoiselle tarkastajalle Tommi Mikkoselle, joka jaksoi antaa palautetta kerta toisensa jälkeen ja ohjasi minua oikeaan suuntaan. Lisäksi haluan esittää kiitokset työni oikolukijoille, äidilleni Leena-Marjalle sekä Noora Pirttilahdelle.

Paljon kiitoksia myös perheelleni tuesta työn tekemisen aikana: äidilleni, isälleni Jarmolle, veljelleni Joonakselle ja hänen avovaimolleen Minnalle. Suuri kiitos myös ystävilleni, joiden kanssa olen saanut jakaa ilot ja surut opiskeluvuosien aikana. Toivottavasti matkamme jatkuu yhdessä myös tulevaisuudessa.

Kuitenkin suurin kiitos kuuluu avopuolisolleni Mikko Airaksiselle. Jaksoit kerta toisensa jälkeen auttaa ja kuunnella. Työni tuskin olisi valmistunut ilman kannustustasi ja arvokasta kritiikkiäsi.

Tampereella 20.5.2015

Karoliina Jäspi

SISÄLLYSLUETTELO

1. JOHDANTO

Web-sovellukset ovat kehittyneet viime vuosien aikana nopeasti. Entiset web-sivustot ovat muuttuneet web-sovelluksiksi, kun niiden sisältö on kehittynyt dynaamiseksi. Webin kehittyessä ovat kehittyneet myös kehitystyökalut ja -kielet sekä testauskäytännöt. Web-sovellukset ovat monimuotoisia, ja niiden laatuvaatimukset poikkeavat osaltaan perinteisistä työpöytäsovelluksista. Erityisesti web-sovellusten testaaminen on osoittautunut suureksi haasteeksi Web-sovellusten kehitystyössä.

Tässä diplomityössä käsitellään testausstrategian luontia olemassa olevaan web-sovellukseen. Työn soveltava osuus on tehty tamperelaiselle yritykselle, joka toimii riskienhallinnan alalla. Diplomityön kohderyhmänä ovat web-sovellusalan toimijat niin kehitys- kuin testauspuolellakin. Työssä oletetaan lukijalta perustietoa perinteisten ohjelmistosovellusten kehittämisestä sekä ymmärrystä ohjelmistoalasta yleisesti.

Kohdeyrityksessä on vuodesta 2005 kehitetty riskienhallintatyökalua, jonka testausprosessi ei ole vakiintunut. Yrityksen riskienhallintatyökalu on tyypillinen web-sovellus asiakas – palvelin -arkkitehtuurimallilla. Yrityksellä on tarve kokonaisvaltaiselle testausstrategialle, jolla yrityksen järjestelmän laatua saataisiin parannettua. Testausstrategiaa luotaessa tulee ottaa huomioon, että kohdejärjestelmä oli jo olemassa oleva, julkaistu järjestelmä, johon tehdään kolme kertaa vuodessa versiopäivitys. Versiopäivityksessä julkaistaan uusia ominaisuuksia ja *bugien* eli virheiden ja vikatoiminnallisuuksien korjauksia. Lisäksi yritys julkaisee tarvittaessa pikakorjausjulkaisuja, joilla korjataan vakavia ongelmia toiminnallisuuksissa.

Ongelmina testausstrategian luomisessa ovat yrityksen rajalliset resurssit testaukseen ja web-sovelluksen pitkä kehityshistoria ilman testauskäytäntöjä. Uusien testausmenetelmien tuominen valmiina olevaan web-sovellukseen on haastavaa, ja sopivien työkalujen löytäminen ei ole itsestäänselvyys. Yrityksessä on myös pitkät perinteet sovelluksen kehittämisestä ja siitä johtuen uusien prosessien tulee olla sellaisia, että kehitystiimi ymmärtää niiden tarpeellisuuden ja on valmis sitoutumaan niihin.

Luvussa 2 esitellään web-sovellusten tyypillisiä ominaisuuksia ja erityispiirteitä verrattuna perinteisiin sovelluksiin. Luvussa 3 käsitellään web-sovellusten testaamista ja miten web-sovellusten testaaminen eroaa perinteisten sovellusten testaamisesta. Luvussa 4 esitellään kohdejärjestelmä, käydään läpi testausstrategian lähtökohdat sekä luodaan teorian pohjalta testausstrategia. Luvussa 5 tarkastellaan kahden eri versiojulkaisun avulla miten testausstrategia toimi käytännössä osana yrityksen kehitys- ja testausprosessia ja mitä muutoksia testausprosessiin tehtiin versiojulkaisujen testaamisen perusteella opituista

asioista. Luvussa 6 arvioidaan testausstrategiaa kokonaisuutena ja esitetään jatkokehitysideat testausstrategialle ja yritykselle. Lopuksi luvussa 7 tehdään yhteenveto työstä ja sen tuloksista.

2. WEB SOVELLUSYMPÄRISTÖNÄ

Web-sovelluksella tarkoitetaan järjestelmää, jota käyttäjä käyttää web-selaimellaan internetin kautta (Dobolyi, 2010). Web-sovellus eroaa perinteisimmistä web-sivustoista dynaamisen sisältönsä takia. On yleistä, että web-sovellusten yhteydessä käytetään termiä ”Web 2.0”. Tällä termillä viitataan muutokseen, joka web-sovelluksissa tapahtui 2000-luvun ensimmäisen vuosikymmenen puolessa välissä. Web-sivustot olivat ennen pääosin toisiinsa linkitettyjä tekstisivuja, mutta muutoksen myötä web-sivustoille tuli enemmän toiminnallisuutta. (O’Reilly, 2007) Kehittyneitä web-sovelluksia voidaan kutsua myös jossain yhteydessä nimellä *Rich Internet Applications* eli lyhennettynä RIA (Taivalsaari *et al.*, 2008).

Erilaisia web-sovelluksia on paljon, ja niiden tarkoitus, kohderyhmä sekä toteutusteknologiat vaihtelevat. Tässä luvussa esitellään web-sovellusten kehitystyön erityispiirteitä ja millaisia haasteita web-sovellukset asettavat kehitystyölleen.

2.1 Web-sovellusten luonne

Alkuaikojen web-sivustoja voidaan kuvailla staattisiksi sivustoiksi. Staattiset sivut ovat tiedostoja palvelimella, eikä niillä ole muuttuvaa sisältöä, kuten esimerkiksi kelloa. Dynaaminen sivu on taas sivu, jolla on muuttuvaa sisältöä. Dynaamiset sivut voivat sisältää muuttuvaa tietoa, joka luodaan vasta sivua ladattaessa ja voi olla esimerkiksi käyttäjästä riippuvaista. Dynaamista sisältöä ovat esimerkiksi tietokannasta haettavat tiedot käyttäjän haun perusteella ja niiden esittäminen. Nykyään web-sovellukset muistuttavat yhä enemmän perinteisiä työpöytäsovelluksia monimutkaisuutensa ja dynaamisen luonteensa vuoksi. (EDInteractive, 2015)

Web-sovelluksissa on paljon ulkoisia tekijöitä, jotka eivät liity web-sovelluksen toteutusteknologioihin. Tällaisia ulkoisia tekijöitä ovat esimerkiksi web-sovellusten käyttäjät. Web-sovelluksilla on hieman erilaisia laatuvaatimuksia kuin perinteisillä sovelluksilla, ja web julkaisualustana tuo myös uudenlaisia haasteita web-sovelluksille perinteisten työpöytäsovellusten haasteisiin verrattuna.

Web-sovelluksille on tyypillistä käyttäjien monimuotoisuus ja suuri määrä. Sovelluksella voi olla käyttäjiä ympäri maailmaa. Heillä on erilaiset kulttuuritaustat, ja heillä voi olla hyvin erilaiset käyttötaidot ja sovellustottumukset. Erilaiset käyttötottumukset ovatkin web-sovellusten kehittäjien haasteena. Perinteisistä sovelluksista voitiin tehdä helpommin lokalisoituja. (Pan *et al.*, 2012)

Ennen web-sivustojen käyttäjiin viitattiin vierailijoina, mutta nykyään web-sovellusten käyttäjiin viitataan käyttäjä-termillä, mikä kuvastaa sitä, miten ihmisen rooli on muuttunut web-sivujen kehittyessä web-sovelluksiksi (Offut, 2002). Myös asiakas-termillä viitataan web-sovellusten käyttäjiin. Tämä voi johtua esimerkiksi siitä, että verkkokauppojen määrä internetissä on lisääntynyt huomattavasti. Verkkokaupoissa käyttäjän rooli asiakkaana on selkeä.

Web-sovelluksia voidaan käyttää monilla eri alustoilla. Käyttäjillä on hyvin erilaisia tietokoneita, joissa on käytössä eri käyttöjärjestelmiä ja selaimia. Mobiililaitteiden osuus internetin käytön päätelaitteina on lisääntynyt viime aikoina. Web-sovelluksia kehittäessä tulee ottaa huomioon, miten web-sovellus toimii eri päätelaitteilla. Usein pyritään siihen, että web-sovellus olisi käytettävissä kaikilla laitteilla, joita asiakkailla on käytössä. (Offutt, 2002)

Web-sovellusten laatuominaisuudet perinteisten sovellusten laatuominaisuuksiin verrattuna ovat erilaiset. Tutkimusten mukaan web-sovellusten tärkeimpinä laatuominaisuuksina pidetään luotettavuutta, käytettävyyttä ja tietoturvallisuutta. Muita web-sovelluksissa tärkeitä laatuominaisuuksia ovat saatavuus, skaalautuvuus, ylläpidettävyys ja markkinoilletuontiaika. (Offutt, 2002) Web-sovellusten kehittäjät arvioivat itse että web-sovellusten luotettavuus, käytettävyys ja turvallisuus ovat tärkeimpiä laatuvaatimuksia (Offutt, 2002).

Web-sovellusten tarjoajille on tärkeää, että asiakkaat palaavat käyttämään heidän sovellustaan, eivätkä vaihda kilpailijan tuotteeseen. Web-sovellukset ovat helpommin saatavilla kuin perinteiset sovellukset, joten käyttäjän on myös helpompi vaihtaa käyttämäänsä web-sovellusta. Tämä asettaa erityisiä vaatimuksia tuotteen laadulle. (Offutt, 2002) Tästä syystä myös käytettävyyden merkitys web-sovelluksissa korostuu.

Perinteisistä sovelluksista web-sovellukset eroavat erityisesti sovelluksen toimituksessa. Perinteiset sovellukset piti toimittaa asiakkaalle, jotta tämä voisi asentaa ne tietokoneelleen. Web-sovellusten jakelu taas on hyvin erilaista: kehittäjät julkaisevat internetissä version ja pystyvät toimittamaan siihen nopeasti korjauksia mikäli on tarpeen. Web-sovellusten julkaiseminen on kuluiltaan edullisempaa eikä niiden toimittaminen vaadi samanlaista ketjua kääntämisestä asentamiseen kuin perinteiset työpöytäsovellukset. Uudet versiot web-sovelluksista pystytään julkaisemaan hyvin nopeasti, jopa minuuteissa. Vanha julkaisuprosessi on myös kallis web-sovelluksen päivitykseen verrattuna. (Mikkonen *et al.*, 2010)

Loppukäyttäjiltä web-sovelluksen käyttöönotto ei myöskään vaadi samanlaista prosessia kuin ennen. On myös mahdollista, että tietyissä tilanteissa käyttäjät eivät välttämättä huomaa mitenkään web-sovelluksen päivittymistä. Tällainen muutos mahdollistaa myös erilaisen kehitysmetodologian web-sovellusten kehitystyöhön. Lyhyet iteraatiot kehityk-

sessä ja niin kutsutut agile-menetelmät sopivat hyvin web-sovellusten julkaisutapaan. Perinteisissä sovelluksissa julkaisuväli voi olla kuukausia tai jopa vuosia, kun taas web-sovellukset voivat olla jatkuvan päivityksen alla, ja päivityksiä voi tapahtua jopa päivittäin. (Jazayeri, 2007)

2.2 Web-sovellusten tekninen toteutus

Web-kehityksen alussa sovellusten toteutustekniikoita ei kehitetty vastaamaan web-sovellusten tarpeita. Web-teknologiat on pääosin kehitetty luomaan dokumenttiorientuneita web-sivustoja eikä niitä ole tarkoitettu luomaan uudenlaisia web-sovelluksia. Tämä aiheuttaakin ongelmia web-sovellusten toteuttamiseen hyvinä pidettyjen toteutuskäytäntöjen kuten esimerkiksi yksinkertaisuuden, siirrettävyyden ja uudelleenkäytettävyyden kannalta. Tässä kohdassa on käsitelty joitakin web-sovellusten tekniseen toteutukseen liittyviä erityispiirteitä. (Mikkonen *et al.*, 2007)

Web-sovellusten toteutuksessa on tärkeää ottaa huomioon se, että web-sovellus jakautuu kahteen eri osaan, palvelinpään toteutukseen sekä sovelluspään toteutukseen. Usein yhdessä web-sovelluksessa käytetään useita eri toteutuskieliä, joilla on omat erikoispiirteensä.

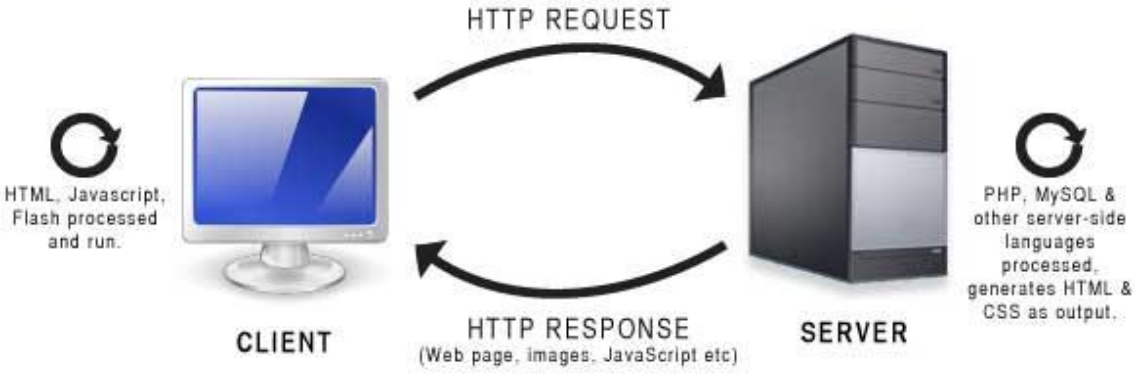
2.2.1 Web-sovellusten arkkitehtuuri

Sovellusten ohjelmistoarkkitehtuurilla kuvataan sitä, mitä osia järjestelmässä on ja millaiset suhteet niillä on keskenään. Ohjelmistoarkkitehtuuria voidaan ajatella järjestelmän perusorganisaationa. (Koskimies, 2005) Usein sovelluksissa voidaan soveltaa useampia arkkitehtuuriratkaisuita.

Web-sovellukset noudattavat pääosin client–server- eli asiakas–palvelin -arkkitehtuurimallia. Tässä mallissa sovellus jaotellaan asiakkaan ja palvelimen osaan. Asiakaspään sovellus tekee pyyntöjä palvelinpäälle, joka käsittelee asiakaspään pyynnöt ja palauttaa asiakaspäälle pyynnön mukaisesti dataa. Kuvassa 1 on esitetty asiakas–palvelin -arkkitehtuurimallin kuva, jossa *client* on asiakas eli selain ja *server* on palvelin. Selain käsittelee tekstitiedostoa, jonka palvelin tuottaa, ja muodostaa tekstitiedoston perusteella käyttäjälle näkymän. Palvelin on yhteydessä tietokantaan ja muodostaa selaimen pyyntöjen mukaisesti halutun tekstitiedoston selaimelle palvelinpään toteutuskielillä.

Toinen yleisesti hyvänä pidetty kerrosarkkitehtuurimalli, jota voidaan käyttää ja soveltaa web-sovelluksiin on MVC-malli. Tässä mallissa näkymä, toteutus ja datamalli erotellaan omiksi kerroksikseen. Tästä on etuja tiedon käsittelyyn ja erilaisten näkymien toteutukseen. Riippuvuudet ovat pieniä, sovelluksen kehitys isommissa järjestelmissä nopeutuu ja muutosten tekeminen on helpompaa. MVC-mallin huomattiin vastaavan dynaamisten web-sovellusten kehitystyössä olevaan ongelmaan siitä, että web-sovellusten lähdekoodissa eri osat sekoittuivat ja järjestelmän rakenne oli sekava. Onkin todettu, että MVC-

kerrosmallin toimivan web-sovellusten kehityksessä hyvin (Jazayeri, 2007). MVC-mallissa on ongelmana se, että se voi lisätä sovelluksen toteutuksen monimutkaisuutta tarpeettomasti. MVC-mallin käyttäminen pienempiin sovelluksiin voi olla turhaa. (Leff, *et al.,* 2001)



Kuva 1. *Asiakas–palvelin -arkkitehtuurimalli kuvattuna (Storey, 2015)*

Vaikka MVC-mallin todetaan sopivan web-sovellusten suunnitteluun, on sen soveltaminen web-sovelluksiin käytännössä hieman ongelmallista. Suurin ongelma on web-sovelluksen hajautettu luonne, jossa asiakas sijaitsee fyysisesti eri paikassa kuin palvelin. Näin ollen sovellus jakautuu sekä konseptuaalisiin kerroksiin (MVC) että fyysisiin tasoihin (Web-ohjelmointi, 2015). Yksi ratkaisu on hyödyntää mallia kokonaan palvelinpään MVC-mallina. (Morales-Chapparo *et al.,* 2007). Tällöin kaikki kerrokset sijaitsevat palvelimen tasolla. Tästä toteutuksesta käytetään joskus nimeä kevyt asiakas, ja sen etuina mainitaan muun muassa se, että selainlaitteelta ei vaadita niin paljoa suorituskykyä. Kevyt asiakas -ratkaisua hyödyntävät sivut on myös helpompi hakukoneoptimoida. Toisessa ratkaisussa, niin sanotussa raskaan asiakkaan mallissa, näkymä ja toteutus sijoitetaan selaimeen, jolloin kokonaisia sivuja tarvitsee ladata palvelimelta harvemmin. Tätä ratkaisua käyttämällä sivusta voidaan ladata vain päivitystä tarvitseva osa uudestaan ilman, että koko sivu pitää ladata uudelleen. Tähän voidaan käyttää AJAX-protokollaa. (Web-ohjelmointi, 2015) AJAX on lyhenne sanoista Asynchronous JavaScript and XML (W3Schools, 2015). Kolmas mahdollinen ratkaisu on niin sanottu dual-MVC-malli, jossa sekä palvelinpään että asiakaspää on jaoteltu sisäisesti MVC-mallin mukaisiin kerroksiin (Leff *et al.,* 2001).

MVC-mallia on siis mahdollista hyödyntää web-sovelluksissa, mutta tällöin tulee harkita tarkkaan millä tavalla mallia sovelletaan ja millaisiin osiin web-sovellus jaetaan. MVC-malli pakottaakin kehittäjän tekemään ratkaisuja ohjelman rakenteesta etukäteen hyvinkin tarkasti. Päätösten tekeminen niin tarkalla tasolla, kuin mallin hyödyntäminen vaatii, voi kuitenkin kehityksen alkuvaiheessa olla erittäin haastavaa. Teknologiavalinnat vaikuttavat siihen, millaista jakoa sovelluksen rakenteesta voidaan tehdä, ja joskus teknologiat eivät taivu parhaimpaan jaottelumalliin. (Leff *et al.,* 2001)

2.2.2 Web-sovellusten asiakaspään toteutusteknologioita

Web-sovellusten luomiseen on monia eri toteutusteknologioita. Tässä kohdassa on käsitelty niistä muutama tyypillisin. Web-sovellukset toteutetaan yhdistelemällä eri toteutusteknologioita, koska esimerkiksi palvelinpäällä ja asiakaspäällä on erilaiset tarpeet toteutuskielille. Myös asiakaspäässä yhdistellään useampaa toteutusteknologiaa.

Sivujen ulkoasua ja rakennetta määritellään HTML-kielellä. HTML eli *Hyper Text Markup Language* on merkkauskieli. HTML on universaali kieli, josta tällä hetkellä käyttöön on tullut HTML 5 -versio. Toteutuksellisesti HTML-sivut ovat vain tekstitiedostoja. Selain osaa tulkita näitä tekstitiedostoja ja luo niistä käyttäjille näkyvät sivut. HTML tukee skriptikieliä, joilla voidaan toteuttaa toiminnallisuutta HTML-dokumentteihin. (Brooks, 2011)

CSS eli *Cascading Style Sheet* on W3C standardoitu merkkaustapa, jolla voidaan määritellä, miltä HTML-sivun tulee näyttää. CSS:ää käyttämällä web-sivuston ulkoasu voidaan erottaa toiminnallisuudesta web-sivujen suunnittelussa. On hyödyllistä, että sivuston ulkoasun määrittävä koodi voidaan erottaa omaksi tiedostokseen ja lähdekoodista saadaan luettavampaa. (Grannel, 2007)

JavaScript on dynaaminen oliokieli, jolla kehitetään erityisesti web-sovelluksia. JavaScript täydentää HTML-dokumentteja. JavaScriptiä tai vastaavaa tarvitaan esimerkiksi vuorovaikutukseen käyttäjän kanssa. (Brooks, 2011)

JavaScriptillä on maine kömpelönä kielenä, jolla ei ole tukea kaikilla selaimilla ja jonka virheiden jäljitys on vaikeaa. Tätä varten on luotu kirjastoja, jotka helpottavat kielen käyttöä. Kirjastojen hyötynä on myös nopeampi kehitysnopeus niiden syntaksin selkeyden ja opittavuuden takia. Kirjastot tukevat erityisesti ajon aikana tapahtuvaa toimintaa ja Ajax-tekniikkaa. (Lengstorf, 2010)

2.2.3 Web-sovellusten palvelinpään toteutusteknologioita

Palvelinpään arkkitehtuuri koostuu yleensä ulospäin avoimesta HTTP-rajapinnasta, palvelimen sisäisestä logiikasta ja tietokannasta. Asiakas tekee pyyntöjä palvelimen HTTP-rajapinnalle, ja palvelin palauttaa asiakkaan pyytämät resurssit. Kuten luvussa 2.2.1 esiteltiin, palvelimen sisäisen logiikan määrä riippuu siitä, kuinka sovelluksen konseptuaaliset kerrokset jaetaan asiakkaan ja palvelimen välille.

Yksi yleinen kieli web-sivujen toteutukseen palvelinpäässä on tällä hetkellä PHP. Se on suunniteltu erityisesti web-palvelinympäristön kehityskieleksi (Artzi *et al.*, 2008). PHP generoi palvelinpäässä koodia, jonka palvelin sitten lähettää asiakaspäälle esitettäväksi.

Kaikki PHP-koodi suoritetaan palvelimen päässä ja selaimelle lähetään vain valmis tekstitiedosto. Käyttäjä ei pääse tarkastelemaan PHP-kielellä web-sovellukseen toteutettuja ominaisuuksia. (Storey, 2015)

Muita palvelinpään toteutuskieliä ovat esimerkiksi Node.js ja Python-Web. Näistä kolmesta toteutuskielestä PHP pärjää tehokkuudeltaan parhaiten kun pyyntöjä on vähän, mutta Node.js on selvästi tehokkaampi rinnakkaisuutta vaativissa tilanteissa. PHP:ta pidetään helpompana kehitystyökaluna kuin asynkronista Node.js:ää. PHP:lla on myös pidemmät perinteet kehityskielenä, ja pitkän kehityshistorian myötä monia ongelmakohtia kielessä on saatu parannettua. (Lei, 2014)

Näiden palvelinpään toteutuskielten lisäksi mainitsemisen arvoinen kieli on Java, joka on suosittu erityisesti suuriliikenteisten sivustojen toteutuksessa. Esimerkiksi erittäin suositut verkkosivujen Linkedin.com:in ja Ebay.com:in palvelinpää on toteutettu Javalla. (W3Techs, 2015)

2.2.4 Web-sovellusten tekninen toteutus käytännössä

Web-sovellusten lähdekoodia tarkasteltaessa voidaan todeta yleisellä tasolla niiden olevan hyvin sekavia. Niitä kuvaillaan joskus spagettikoodiksi, jolla tarkoitetaan, etteivät ne etene järjestelmällisesti ja että niissä voidaan käyttää paljon perinteisten sovellusten kehitystyössä huonoina käytäntöinä pidettyjä ratkaisuja. Näitä ovat esimerkiksi globaalit muuttujat ja goto-käskyt. (Mikkonen *et al.,* 2007*)*

Yksi sovellusten kehitystyön perusperiaatteista on johdonmukaisuus. Tämän perusperiaatteen toteutuminen on ongelmana web-sovellusten toteutuskielissä, koska ne voivat tarjota saman toiminnallisuuden toteuttamiseen useamman tavan, ja samassa sovelluksessa voidaankin hyödyntää useita eri tapoja. Tästä syntyy ongelmia esimerkiksi ylläpidettävyydessä ja koodin hallittavuudessa. (Mikkonen *et al.,* 2007*)*

Toinen hyvänä pidetty periaate on sovelluksen osien uudelleenkäytettävyys. Web-sovellusten lähdekoodi on sekavaa ja siinä sekoittuu useita eri teknologioita ja toteutustapoja. Tämä vaikeuttaa koodin uudelleenkäytettävyyttä. Lisäksi web-sovellukset saattavat sisältää paljon arvoja, jotka on upotettu lähdekoodiin, mikä myös vaikeuttaa koodin uudelleen käyttöä. (Mikkonen, *et al.,* 2007*)*

Web-sovellusten kehitystyössä ongelmana on, että kielet ovat yleensä tulkattavia. Tästä aiheutuu se, ettei kehittäjä voi varmistua etukäteen ovatko kaikki toiminnallisuudet toteutettu ja mitä puutteita koodissa on. Tämän lisäksi sovellus voi ajon aikana muuttaa itseään, mistä aiheutuu useampia käyttötapauksia, joiden toimimisen varmistaminen on haastavaa. Näistä johtuen web-sovellusten testaukseen tuleekin kiinnittää paljon huomiota. (Mikkonen, *et al.* 2007*)*

Web-kehityksessä usein käytetyt JavaScript ja PHP ovat hyvin sallivia kieliä ja kehittäjä voi tehdä suuria virheitä koodatessaan niillä ilman että saa virheilmoitusta sovellusta ajaessaan. Tämä on eräs seikka, joka asettaa haasteita web-sovellusten testaamiselle. JavaScript ei esimerkiksi ilmoita ennen ajoa, jos jokin asia, johon on viitattu, puuttuu lähdekoodista. Kehitystyötä tulisi tehdä vähitellen ja jatkuvasti koodia testaten, jotta ongelmat löydettäisiin nopeammin. Suuremman koodimäärän testaaminen kerralla on hidasta, kun selviä virheilmoituksia ei ole. (Taivalsaari *et al.,* 2008*)*

2.3 Selaimen toiminnallisuudet

Yksi ero perinteisten sovellusten ja web-sovellusten välillä on se, että web-sovellusta voidaan käyttää selaimen toiminnallisuuksien kautta eli itse sovelluksen ulkopuolelta. Selain tarjoaa käyttäjälle esimeriksi navigaatiotoiminnallisuuksia kuten ”takaisin”, ”eteenpäin” ja ”päivitä”-toiminnallisuudet (Zhu *et al.*, 2009). Nämä toiminnot toimivat hyvin staattisten web-sivustojen kanssa, mutta dynaamisten web-sovellusten kanssa ne aiheuttavat ongelmia. Esimerkiksi pankkitoiminnot ovat sellaisia toimintoja, joissa ”takaisin”-ominaisuutta ei tietyissä tapauksissa pitäisi sallia. Jos kehitystyössä ei ole otettu huomioon tällaista tilannetta, voi aiheutua vakavia virhetiloja, jotka voivat esimerkiksi vaarantaa käyttäjän omaisuuden.

Web-sovellusten suunnittelussa onkin ongelmana selaimen tarjoamien palveluiden huomioiminen. Web-sovellusten kehittäjien tulee ottaa huomioon selaimen tarjoamat toiminnallisuudet suunnittelutyössä ja pyrkiä varmistumaan, että sovelluksen oikeanlainen toiminta jatkuu myös käyttäjän käyttäessä selaimen toiminnallisuuksia. Tulevaisuudessa voi olla tarpeellista, että selaimen toiminnallisuudet pystytään laittamaan pois päältä ongelmia aiheuttavissa tilanteissa.

2.4 Web-sovellusten tietoturva

Web-sovelluksissa tietoturvan merkitys korostuu suuresti. Tämä johtuu siitä, että sovellukset ovat laajemmin kaikkien saatavilla – myös vihamielisten käyttäjien. Erilaiset tietoturva-aukot mahdollistavat hyökkääjien pääsyn järjestelmään. Hyökkääjät saattavat päästä muokkaamaan web-sovellusta tai hyödyntämään sovelluksen tietokannoissa olevia salaisia tietoja.

Web-sivuston ylläpitäjät ovat vastuussa käyttäjilleen siitä, etteivät käyttäjän tiedot paljastu ulkopuoliselle taholle. Erityisesti tämä tarve korostuu, kun käsitellään rahaa tai jotain arkaluontoista tietoa, kuten esimerkiksi potilastietoja. Myös vähemmän arkaluontoista tietoa sisältävien sivustojen ylläpitäjien tulee kiinnittää huomiota tietoturvaan, etteivät käyttäjien käyttäjätiedot ja salasanat paljastu ulkopuoliselle. Useat käyttäjät saattavat käyttää samaa salasanaa tai sen variaatioita muissa palveluissa, vaikka tämä ei ole suositeltua ja tietoturvallista. Hyökkäyksistä voi aiheutua myös maineen menetystä ja taloudellisia tappioita.

SQL-injektio on yksi yleisimpiä tietoturvahyökkäyksiä. Monet web-sovellukset hyödyntävät tietokantoja erilaisten tietojen säilömiseen. Esimerkiksi käyttäjätunnukset ja salasanat säilötään tietokannassa. Web-sovellusten suuri tietoturvariski liittyy tietokantoja vastaan tehtyihin hyökkäyksiin. Tällaisessa hyökkäyksessä hyökkääjä saa syötettyä web-sovelluksen tietokantaan suoraan vaikuttavia SQL-komentoja tietokantapalvelimelle. Jo web-sovellusten kehitysvaiheessa on syytä varmistua, että hyökkääjille ei jätetä mahdollisuutta tämän tyyppiseen hyökkäykseen. Web-sovellusten testauksessa tulee varmistua, että sovellusta testataan tällaista uhkaa vastaan. (Acunetix, 2015)

Cross site scripting on yksi tietoturva-aukkotyyppi, joita web-sovelluksissa on. Tätä hyödyntämällä hyökkääjä pystyy muokkaamaan koodia ja esittämään sitä tarkoituksenmukaisena eli ilman, että käyttäjä tietää käyttävänsä vihamielisesti manipuloitua sivua. Hyökkääjä voi kerätä arkaluontoista dataa, vaihtaa käyttäjäasetuksia tai esimerkiksi muokata käyttäjän evästeitä. Hyökkääjä voi ujuttaa haitallisen koodin vain yhden käyttäjän selaimeen tai koko sivuston palvelimelle. Cross site scripiting -haavoittuvuuksia on ollut useilla hyvin tunnetuilla sivustoilla. Tapauksia löytää helposti internetin hakukoneiden avulla. Web-sovelluksia kehitettäessä tätä hyökkäystä voidaan estää käyttämällä HTML-koodin siistimistä. (CGISecurity, 2002)

Kuten aiemmin tässä työssä mainittiin, selaimet tarjoavat toiminallisuuksia, joilla pystytään vaikuttamaan web-sovellukseen. Erilaisten liikkumis- ja päivitystoimintojen lisäksi selaimet tarjoavat mahdollisuuden tarkastella sivun lähdekoodia. Tämä tarkoittaa, että kuka tahansa pystyy tarkastelemaan miten sivusto on toteutettu. Tästä aiheutuu tietoturvaongelma, mikäli tarvittavia koodin toiminnallisuuden piilottamistoimintoja ei tehdä.

Kirjautumistunnukset ovat erityisen arkaluontoista tietoa, kuten aiemmin työssä mainittiin. On tärkeää, että käyttäjän pyytäessä uutta salasanaa, hänelle ei lähetetä hänen vanhaa salasanaansa, vaan hänet pakotetaan linkin kautta vaihtamaan itselleen uusi salasana. Käyttäjä on saattanut asettaa itselleen salasanan, jota käyttää muissakin järjestelmissä, ja tällöin tämä salasana toimitettaisiin hyökkääjien saataville. Tässä tulee huomioida, että salasana tulee uusia vasta, kun sähköpostiin tullutta linkkiä painetaan, jotta käyttäjille ei pystytä tekemään ilkivaltaa nollaamalla heidän salasanojaan. Muutenkaan käyttäjien salasanoja ei tulisi tallentaa tietokantaan suojaamattomana. Tästä voi aiheutua käyttäjille vakavia seuraamuksia, jos tietokannan raakadata joutuu väärin käsiin.

Ennen web-sivustoilla käytettiin paljon turvakysymyksiä, joilla varmistettiin käyttäjän henkilöllisyys. Nykyään kuitenkin internetin kautta ihmisistä löytyy niin paljon tietoa, että tietoturvakysymykset ovat menettäneet merkitystään. Tämän tilalle on tullut 2-vaiheinen tunnistautuminen, jossa käyttäjän pitää varmistaa kirjautumisensa tai salasananvaihtonsa esimerkiksi puhelimensa tai toisen sähköpostinsa kautta. Tällöin hyökkääjällä pitää olla pääsy molempiin tunnistuskeinoihin, jotta hän voisi murtautua käyttäjän tunnuksilla järjestelmään.

Erilaisia teknisiä tietoturvahyökkäyksiä vastaan suojautumista on helppo kehittää paremmaksi, kun tiedostetaan järjestelmää toteutettaessa tietoturvallisuuden vaatimukset ja järjestetään asianmukaisen tietoturvatestaus. Kuitenkin suurempi osa web-järjestelmiin tehdyistä hyökkäyksistä tapahtuu *social engineeringing* eli sosiaalisen manipuloinnin avulla. Hyökkääjä saa käyttäjän luovuttamaan hänelle tarpeelliset tiedot järjestelmään sisälle pääsemiseksi. Tällaista uhkaa ei voida tietoturvatestauksella testata, vaan tähän tietoturvariskiin auttaa vain käyttäjien kouluttaminen tiedostamaan tällainen uhka ja toimimaan sitä vastaan. (Winkler, *et al.* 1995)

3. WEB-SOVELLUSTEN TESTAUS

Perinteisten sovellusten testaaminen on haastava tehtävä sekä teknisesti että testausprosessina. Web-sovellukset tuovat entisestään haastavaan testaukseen paljon uusia ongelmia. Erityisesti web-sovellusten tekninen toteutus sekä tietoturvan kaltaiset erityispiirteet aiheuttavat sen, että web-sovellusten testaamisessa pitäisi olla erilaiset käytännöt kuin perinteisten sovellusten testaamisessa. Tässä luvussa käsitellään web-sovellusten testaamista sekä testausstrategian luomista.

3.1 Testausstrategia

Testausstrategialla voidaan tarkoittaa eri asiaa riippuen asiayhteydestä. Tässä työssä testausstrategialla tarkoitetaan suunnitelmaa, jossa on vastattu kysymyksiin mitä testataan, miten testataan ja miksi testataan.

Testausstrategian luomisen voidaan ajatella jakautuvan kolmeen eri vaiheeseen: vaatimusten määrittelemiseen, riskiarviointiin ja prioriteettien määrittelemiseen sekä varsinaiseen testausstrategian luomiseen. Ensimmäisessä vaiheessa asetetaan testausstrategialle tavoitteet. Tässä vaiheessa käydään läpi millaisia vaatimuksia järjestelmälle on asetettu. Toisessa vaiheessa arvioidaan, mitkä vaatimuksista ovat erityisen kriittisiä järjestelmälle ja mihin testaamisessa tulisi panostaa. Kolmannessa vaiheessa luodaan varsinainen testausstrategia, eli valitaan mitä testataan ja miten testataan. (Zemin *et al.*, 2011)

Testausstrategiassa on tärkeää, että se on monipuolinen. Eri testausmenetelmien yhdisteleminen on tarpeellista luotaessa kattavaa testausstrategiaa. Testausstrategiaan ei kuulu ainoastaan työkalujen ja lähestymistapojen valinta, vaan siihen kuuluu myös se, miten testausprosessia hallitaan: kuka tekee mitä ja milloin testit suoritetaan. Lisäksi testausstrategiassa tulee määritellä, miten testaamisesta raportoidaan ja miten relevantit tulokset esitetään. (Zemin *et al.*, 2011) Testausstrategiassa käydään läpi, miten toiminnallinen testaus toteutetaan eri tasoilla ja miten ei-toiminnallista testausta tehdään. Web-sovelluksissa testausstrategian pitäisi kattaa ainakin seuraavat osa-alueet: toiminnallinen testaus, käytettävyystestaus, käyttöliittymätestaus, yhteensopivuustestaus, suorituskykytestaus ja tietoturvatestaus. (Software Testing Help, 2015) Näiden lisäksi voidaan myös arvioida, tulisiko järjestelmälle tehdä saavutettavuustestausta.

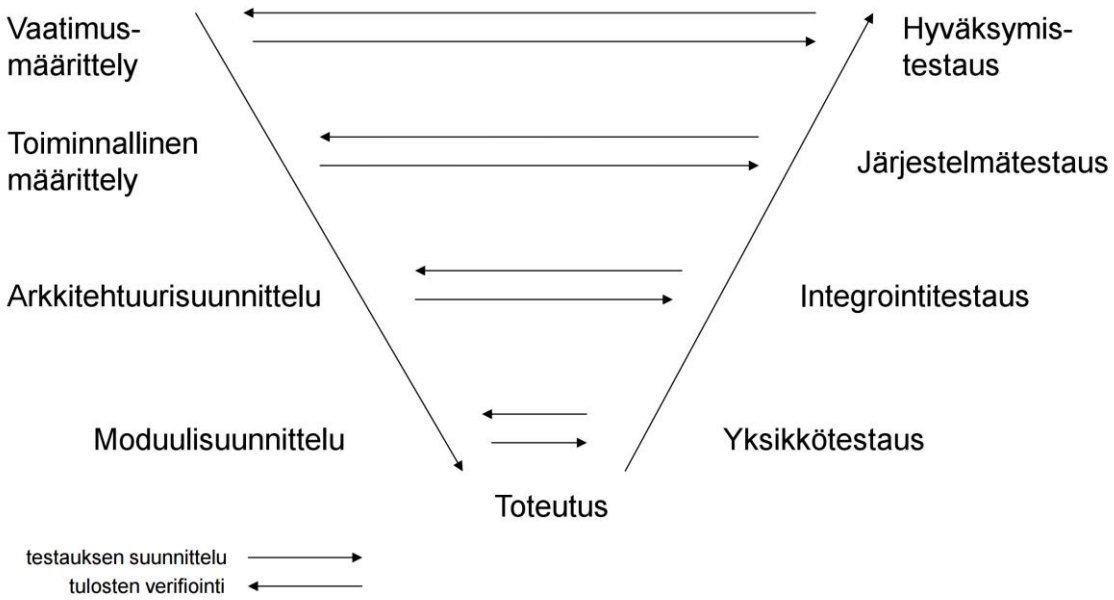
Testausstrategian luomiseen vaikuttaa se, luodaanko testausstrategia jo valmiiseen sovellukseen vai kokonaan uuteen sovellukseen. Osa testausmenetelmistä tulee ottaa alusta asti käyttöön, jos ne halutaan mukaan testaukseen. Joidenkin testausmenetelmien myöhempi implementoiminen järjestelmään voi viedä yhtä paljon resursseja kuin järjestelmän kehitys alun perin. Olemassa olevien sovellusten virheitä korjatessa tulisi ottaa huomioon, että muutokset koodiin saattavat aiheuttaa ongelmia muualla järjestelmässä. Olemassa

olevien sovellusten kehittäjien tulisi testata muutoksia tehdessään myös muita ominaisuuksia, johon muutos saattaa vaikuttaa. (Vrieze, 2012)

3.2 Testaustasot ja testauksen jaottelu

Tässä kohdassa käydään läpi, millä tavoilla testaamista voidaan jakaa eri tasoille ja miten testausta voidaan jaotella lähestymistapojen mukaan. Vaikka web-sovellusten testaaminen eroaa perinteisten sovellusten testaamisesta, testaustasojen mielletään olevan samoja.

Perinteiseen ohjelmistokehityksen vesiputousmalliin on liitetty tyypillisesti testauksen V-malli. Kuvassa 2 on kuvattu testauksen V-malli. V-mallilla kuvataan mikä testaustaso vastaa mitäkin toteutustasoa. V-mallilla myös osoitetaan missä järjestyksessä testaus tulisi suorittaa. Vaikkei ohjelmistokehityksessä noudatettaisi testauksen V-mallia, niin on järkevää, että testauksen eri tasot pysyvät järjestyksessä. Kaikkia yksiköitä ei kuitenkaan tarvitse testata ennen integraatiotestaamisen aloittamista, vaan ne voivat toimia lomittain kun sopivia kokonaisuuksia saadaan testattavaksi.



Kuva 2. *Testauksen V-malli (Katara et al., 2013)*

Mallilla voidaan erottaa testauksen eri tasot: yksikkötestaustaso, integraatiotestaustaso, järjestelmätestaustaso ja hyväksymistestaustaso. Nämä ovat toiminnallisen testauksen muotoja. Ei-toiminnallista testausta ovat esimerkiksi tietoturvatestaus, suorituskykytestaus ja saavutettavuustestaus. (Dobolyi, 2010)

Testaaminen ja testit voidaan jakaa monella eri tapaa osiin. Tällä tavalla testejä voidaan suunnitella paremmin ja niiden käyttötarkoitus on selvempi käyttäjille. Yksi tapa jaotella testejä on niiden jakaminen positiivisiin ja negatiivisiin testeihin. Näistä positiiviset testit

testaavat, toimiiko koodi oikein. Tällöin oletetaan, että testaamisen kohde saa vain oikeanlaisia syötteitä. Negatiiviset estit puolestaan testaavat miten koodin toimii epätavallisissa tilanteissa ja niissä, joissa se saa vääriä syötteitä. (ISTQB, 2007)

Toinen esimerkki testien jaottelutavoista on jakaa testit valko- ja mustalaatikkotesteihin. Valkolaatikkotestauksessa testaajan käytössä on lähdekoodi ja testaus tehdään koodia vasten. Mustalaatikkotestauksessa käytetään valmista sovellusta tai testaamiseen soveltuvaa osaa sovelluksesta ja testaaja testaa toimiiko sovellus kuten määritelty. Kolmas muoto laatikkotesteihin on harmaalaatikkotestaus, joka on kahden edellisen testaustavan välimuoto. (Khan *et al.*, 2012)

3.3 Toiminnallinen testaus

Toiminnallisella testauksella tarkoitetaan sovelluksen määriteltyjen toiminallisuuksien testaamista. Tässä kohdassa käydään läpi, miten toiminnallista testausta toteutetaan eri testaustasoilla ja mitä testauksen menetelmiä liittyy toiminnalliseen testaukseen.

Toiminnallisessa testauksessa tulee ottaa huomioon, että yleisesti pidetään huonona käytäntönä, että kehittäjät itse testaavat omaa koodiaan ja toteuttamiaan toiminnallisuuksia. Heillä saattaa olla helposti puolueellinen lähestymistapa. Tämä on ongelmallista erityisesti web-sovelluksia kehitettäessä, jolloin kehitystiimi tyypillisesti saattaa olla pieni. Kuitenkin nykyään yleistyneet ketterät menetelmät suosivat sitä, että tiimin sisällä ei ole erillisiä rooleja, vaan tiimi itsenäisesti huolehtii koodin tuottamisesta ja laadusta. (Redouane, 2002)

Toiminnalliseen testaukseen liittyy termi regressiotestaus. Sillä tarkoitetaan, että tehdään samoja testejä uudelleen järjestelmään, jotta saadaan selville, onko jokin jo kerran toiminut toiminnallisuus lakannut toimimasta koodin muokkaamisen tai uuden ominaisuuden implementoimisen jälkeen (Osherove, 2009). Jokaisella testaustasolla voidaan suorittaa ja mahdollisuuksien mukaan tulisi suorittaa regressiotestausta.

Web-sovellusten toiminnallisessa testaamisessa korostuu sovelluksen monimuotoisuus. Yksi sivu voi sisältää monia erilaisia elementtejä, ja tämä on suuri haaste web-sovellusten testaamisessa. Myös sivujen rakenne voi olla hieman sekava ja epätarkka. Web-sovelluksissa käytettävyyden arvioiminen ja samalla loppukäyttäjän kokemuksen arviointi ovat tärkeitä seikkoja testaamisessa ja siksi onkin luontevaa, että testatessa painotetaan järjestelmätestausta asiakasympäristössä. (Xu *et al.*, 2004)

3.3.1 Yksikkötestaus

Yksikkötestaus on jonkin ohjelman yksittäisen moduulin tai komponentin testaamista. Yksikön ei pitäisi olla riippuvainen mistään muusta yksiköstä. Yksikön tunnistaminen

web-sovelluksissa on kuitenkin hieman vaikeampaa kuin perinteisissä sovelluksissa. Yksiköksi voidaan valita esimerkiksi myös yksi sivu, johon voi liittyä useampia elementtejä. Näitä yksittäisiä elementtejä voidaan mieltää myös yksiköiksi. On kuitenkin mahdollista, että samaa yksikköä käytetään useammalla sivulla. (Di Lucca *et al.*, 2002) Näitä muita, oletettavasti pienempiä elementtejä, voivat olla esimerkiksi skriptimoduuli, lomake tai joku muu pieni web-objekti. Yksikkötestauksen haasteena on myös asiakaspään ja palvelinpään huomioinen jaottelussa. (Di Lucca *et al*., 2006)

Perinteisissä sovelluksissa yksikkötestaus on ollut valkolaatikkotestausta, eli yksikkötestit on suoritettu suoraan lähdekoodista tietoisina. Web-sovelluksissa käyttäjän rooli korostuu, ja sen vuoksi testejä kannattaa web-sovellusta testatessa suorittaa enemmän loppukäyttäjän näkökulmasta. (Xu *et al.*, 2004) Tällöin myös yksikkötestausta kannattaa web-sovelluksissa tehdä myös mustalaatikkotestauksena. Erityisesti jos yksiköksi valitaan sivu, niin testaaminen voidaan suorittaa niin mustalaatikko-, valkolaatikko- ja harmaalaatikkotestauksena. (Di Lucca *et al*., 2006).

Yksikkötestauksen kirjallisuudessa nousee tyypillisesti esille termi koodikattavuus. Koodikattavuudella tarkoitetaan kuinka suuri osa koodiriveistä tulee suoritettua testin tai testien aikana (Dooley, 2011). Web-sovelluksia kehitettäessä onkin huomioitava, että jos koodikattavuus ei ole täydet sata prosenttia, voi sovelluksessa olla vakaviakin ongelmia. Kuitenkin vastaavasti on huomioitava, ettei sata prosenttinen koodikattavuus tarkoita, että sovellus olisi virheetön. (Taivalsaari, *et al.*, 2008.*)* Kuten luvussa 2 kuvattiin, dynaamisen luonteensa takia web-sovellusten koodikattavuuden mittaaminen on tärkeää. Koodikattavuudella pystytään saamaan lisää varmuutta siitä, että jokainen mahdollinen toiminnallisuus ja koodirivi on tullut testattua.

Web-sovelluksissa pelkkä koodikattavuus ei riitä varmistamaan yksikön toimivuutta. Web-sovelluksissa tulisi huomioida valkolaatikkotestausta tehtäessä asiakaspään sivulle HTML-kattavuus, web-objektikattavuus, skriptikattavuus, päätöskattavuus sekä hyperlinkkikattavuus. Näiden eri kattavuuksien avulla pystytään tarkistamaan, että jokainen objekti ja hyperlinkki toimii oikein. Palvelinpään valkolaatikkotestauksessa tulisi taas huomioida näiden lisäksi dynaamisesti luotujen sivujen testikattavuus. (Di Lucca *et al.*, 2006).

Yksikkötestauksessa on tärkeää, että testit ovat helposti ja nopeasti uudelleen ajettavissa. Näin voidaan varmistaa, että yksiköt eivät mene rikki jonkin muutoksen yhteydessä. Yksikkötestauksessa regressiotestauksen tuki on tärkeää. (Dooley, 2011) Yksikkötestit nopeuttavat vian löytämistä koodista, koska yksikkötestejä ajamalla selviää suoraan minkä yksikkötestin ajo epäonnistui. Jos yksikkötestejä ei käytetä, voi integraatiotestausvaiheessa olla vaikeampi löytää, missä vika sijaitsee. (Osherove, 2009) Vikojen etsiminen voi kuluttaa paljon resursseja ja aiheuttaa turhaa työtä.

Yksikkötestaus on jopa hieman kiistelty aihe testauksen alalla. On yleistä, että yksikkötestaus jätetään kehittäjien tehtäväksi, jotka eivät ole motivoituneita oman koodinsa testaamiseen. Yksikkötestien laadukkuus on tärkeä osa yksikkötestien onnistumisessa. Yksikkötestit tulee myös suunnitella tarkasti ja testata. Tämä aiheuttaa lisäkuluja ja vaatii resursseja. Lisäksi vaatimusten muuttuessa myös yksikkötestit tulee päivittää, mikä kuluttaa myös resursseja.

3.3.2 Integraatiotestaus

Integraatiotestauksella tarkoitetaan sellaista testausta, jolla testataan sovelluksen eri yksiköiden toimintaa yhdessä. Erilaisia yksiköitä voidaan ajatella viime luvussa esiteltyjen yksiköiden lisäksi olevan esimerkiksi luokat ja oliot. Jos ohjelma on modulaarinen, erilaisten yksiköiden tunnistaminen on helpompaa. Web-sovelluksissa eri yksiköiden tunnistaminen on haastavaa, ja tällöin myös integraatiotestauksen suunnittelussa ja toteutuksessa tulee olla tarkkana.

Web-sovelluksessa integraatiotestauksella voidaan tarkoittaa, miten tietty joukko toisiinsa liittyviä sivuja käyttäytyy yhdessä. Suunnitteludokumentaation tulisi osoittaa, miten eri sivut ovat toisistaan riippuvaisia. (Di Lucca *et al.*, 2006) Yksi tapa on määrittää, että integraatiotestaus tarkoittaa asiakaspään ja palvelinpään sovittamista yhteen. (Di Lucca *et al.*, 2002) Web-sovelluksissa yhdistellään tyypillisesti monia eri toteutusteknologioita, ja tästä syntyy yksi haaste integraatiotestaukseen ohjelman toteutuskielten monimuotoisuuden takia.

Integraatiotestaustasolla tulee huomioida sekä sovelluksen käyttäytyminen että sovelluksen rakenne. Rakenne auttaa määrittelemään mitkä sivut liitetään toisiinsa, ja käyttäytymismalli taas ohjaa miten niiden tulisi toimia yhdessä. Tästä syystä harmaalaatikkotestaus onkin integraatiotestaustasolle sopiva testausmenetelmä. (Di Lucca, *et al.*, 2006)

Yksikkötestien automatisointia voidaan hyödyntää integraatiotestauksessa. Automatisoiduilla testeillä voidaan päästä käyttämään *Continous integration* -menetelmää eli jatkuvaa integraatiota. Jatkuvan integraation menetelmällä kehittäjät yhdistävät tuottamansa toiminnallisuuden usein varsinaiseen koodiin, ja tällöin koodin pitää läpäistä tietyt testit. Näin varmistetaan, ettei lisäys riko mitään jo toimivaa ominaisuutta. Tällä tavoin virheitä aiheuttavaa koodia ei pääse niin helposti lopulliseen järjestelmään ja ongelma ei pääse kasvamaan isommaksi. Läpäisyvaatimuksella saadaan helposti selville, missä ongelma on. Jos tällainen menetelmä ei ole käytössä, voidaan koodia implementoida järjestelmään suuri määrä ja vian aiheuttavan koodipätkän löytäminen on vaikeampaa.

3.3.3 Järjestelmätestaus

Järjestelmätestaustasolla puhutaan web-sovelluksissa koko sovelluksen testauksesta, joka yleensä tapahtuu käyttöliittymätestauksen kautta. Tällä tarkoitetaan käyttöliittymän

kautta tapahtuvaa sovellukselle suoritettavaa testausta (Tian, 2005). On kuitenkin mahdollista hyödyntää myös harmaalaatikkotestaustapoja, koska järjestelmätestaustasolla voidaan mitata esimerkiksi hyperlinkkikattavuutta, jota voidaan suorittaa myös valkolaatikkotestauksella.

Web-sovelluksia, kuten muitakin sovelluksia, on järkevä testata asettuen loppukäyttäjien asemaan (Xu *et al*., 2004). Järjestelmätestauksen merkitys web-sovelluksissa korostuukin ja etenkin mustalaatikkotestauksen merkitys on tärkeä. Tällöin onkin tärkeää tietää, miten järjestelmän oletetaan toimivan. Kattava dokumentaatio järjestelmän toivotusta toiminnasta onkin edellytys, jotta järkevää järjestelmätestausta voidaan suorittaa jonkun muun kuin järjestelmän omistajan näkökulmasta. Omistajalla tarkoitetaan henkilöä, joka päättää järjestelmän toiminnallisuuksista. Yleensä dokumentaatio tulisi olla, koska sen avulla myös kehittäjät tietävät mitä tehdä. Kehittäjien tulisi dokumentoida kehitystyön aikana tehdyt toiminnallisuuksia koskevat ratkaisut testausta varten.

Kuten aiemmilla tasoilla, myös järjestelmätestaustasolla voidaan mitata kattavuuksia. Tällöin valkolaatikkotekniikkaa hyödynnettäessä voidaan asettaa tiettyjä vaatimuksia testauksen tasolle. Näitä voivat olla esimerkiksi sivutesti, eli jokainen sovelluksen sivu tulee olla vierailtuna. Toisessa esimerkissä jokainen hyperlinkki jokaiselta sivulta tulee testattua. Nämä ovat keinoja varmistua, että jokainen sivu, joita järjestelmässä on tarkoitus olla, on olemassa (Ricca *et al*, 2001). Järjestelmätestauksessa on tärkeää varmistua siitä, ettei järjestelmässä ole kuolleita hyperlinkkejä. (Di Lucca *et al*., 2006) Kuolleet hyperlinkit sovelluksen sisällä osoittavat, ettei kaikkia osia ole onnistuneesti toteutettu tai onnistuneesti liitetty yhteen.

Käyttöliittymän kautta tapahtuva testaus on järjestelmätestauksen pääasiallinen testausmenetelmä. Käyttöliittymätestausta tehdessään testaaja käyttää sovellusta sen valmiin käyttöliittymän kautta ja suorittaa erinäisiä testejä järjestelmälle. Testit voivat olla etukäteen kohta kohdalta suunniteltuja testejä tai niillä voi olla pyöreämpi määritelmä kuten ”testaa tuotteen lisääminen järjestelmään”. Jälkimmäisellä tavalla testaajan pitää olla luovempi ja miettiä erilaisia lähestymistapoja ja virhetilanteita, joita käyttäjä voi saada aikaan järjestelmää testatessaan.

Yksi tapa suorittaa käyttöliittymätestausta on tutkiva testaus. Siinä ei ole etukäteen määriteltyjä testitapauksia. Tutkivassa testauksessa testaaja itse ideoi testitapaussession aikana miten testaus etenee ja miten hänen tulisi testata. Testaaja reagoi siihen, miten testattava sovellus toimii, ja soveltaa testejä tuloksien perusteella. Tutkivan testauksen hyötynä on, että sillä pyritään löytämään sellaisia ongelmia, joita kohdejärjestelmässä ei vielä tiedetä olevan. Jotta tutkivaa testausta voidaan käyttää tehokkaasti, tulee testaajalla olla saatavilla riittävästi tietoa testattavasta järjestelmästä. (Itkonen *et al.*, 2013)

Tutkivan testaus osoitetusti on tehokas löytämään virheitä kohdejärjestelmistä. Lisäksi se on resurssitehokasta. (Itkonen *et al*., 2013) Tutkimukset osoittavat, että selvää hyötyä

etukäteen tarkasti suunnitelluista testitapauksista tutkivaan testaukseen verraten ei ole (Prakash *et al.* ,2011; Itkonen *et al.*, 2013). Tutkiva testaus luottaa enemmän testaajan ammattitaitoon suorittaa testit, mutta jättää helposti mahdollisuuden, että taitavakin testaaja voi unohtaa testata jonkin tärkeän ominaisuuden. Tutkivan testauksen tukena voidaankin käyttää muistilistoja lähestymistavoista ja tärkeistä huomioista, joita järjestelmästä tulisi tehdä.

Tutkivaa testausta voidaan suorittaa sessiopohjaisena, jolloin sessiolla voi olla tietty tarkoitus, esimerkiksi laajempi ominaisuus, jota järjestelmästä tutkitaan (Itkonen *et al*., 2013). Tutkivasta testauksesta on hyvä pitää sessiolokia, johon kirjataan löydetyt viat ja session tarkoitus sekä aika. Näin pystytään seuraamaan testaukseen menevää aikaa sekä kartoittamaan sessioiden hyödyllisyyttä.

Myös järjestelmätestaustasolla regressiotestauksen automatisointi on yleensä kannattavaa. Tämä vähentää käsin tehtävää testausta ja vapauttaa resursseja muuhun testaamiseen. Regressiotestaamisella pystytään varmistamaan, että uuden version tullessa järjestelmä edelleen toimii samalla tavalla kuin ennen muutoksia. (Ricca *et al.,* 2001)

Selaimet tarjoavat toiminnallisuuksia web-sovelluksissa liikkumiseen ja sivun päivittämiseen. Järjestelmätestausta tehdessä tuleekin ottaa huomioon käyttötapaukset, joissa käyttäjä käyttää sovellusta selaimen toiminnallisuuksia hyödyntäen. Tämä lisää käyttötapausten määrää huomattavasti. Erityisesti jos web-sovellusta kehitettäessä ei olla otettu huomioon selaimen toiminnallisuutta kattavasti, voivat tällaiset käyttötapaukset paljastaa suuriakin ongelmia sovelluksesta.

3.3.4 Hyväksymistestaus

Hyväksymistestauksella tarkoitetaan niitä testejä, joiden perusteella järjestelmä hyväksytään julkaistavaksi. Hyväksymistestauksella pyritään testaamaan kaikki järjestelmän tärkeimmät ominaisuudet. Järjestelmälle asetetaan tietyt vaatimukset, jotka sen tulee läpäistä, jotta hyväksymistestaus katsotaan onnistuneeksi ja järjestelmä voidaan julkaista. Nämä vaatimukset tulisi muodostaa hyvissä ajoin dokumentaation pohjalta. (Langer, 2012)

Hyväksymistestauksen automatisointi on agile-menetelmien myötä esille tullut asia. Hyväksymistestauksen tekemistä käsin pidetään kalliina, aikaa vievänä ja tylsänä. (Haugset *et al.*, 2008) Työtehtävien tylsyys voi aiheuttaa helposti sen, että niitä myös hieman ylenkatsotaan eikä niihin keskitytä. Tämä saattaa todennäköisesti lisätä virheiden tekemistä työskenneltäessä.

3.4 Ei-toiminnallinen testaus

Ei-toiminnallisella testauksella tarkoitetaan niiden web-sovellusten ominaisuuksien testaamista, jotka eivät ole suoraan toiminnallisuuksia. Näitä ovat esimerkiksi suorituskyky, käytettävyys, tietoturva ja saavutettavuus. Myös yhteensopivuustestauksesta puhutaan ei-toiminnallisen testauksen yhteydessä. (Di Lucca *et al.,* 2006*)*

3.4.1 Tietoturvatestaus

Tietoturvaskannerit ovat ohjelmia, jotka yrittävät toimia hyökkääjän tavoin ja etsivät sovelluksista tai sivustoilta tietoturvaongelmia. Tietoturvaskannerit muodostavat raportin, joista käy ilmi mille uhkille sivusto tai sovellus on altis. Näitä skannereita on tarjolla monia erilaisia. Kuten lähes kaikkia ohjelmia, myös tietoturvaskannereita on sekä vapaan lähdekoodin ohjelmistona että kaupallisina ohjelmistoina. Tietoturvaskannerin valinnassa tulee tunnistaa mitkä uhat ovat vakavimpia sivustolle ja tarkastella useita eri vaihtoehtoja, jotta löytää parhaiten sovelluksen tarpeita vastaavan skannerin. Tietoturvaskannerin valitsemisessa tulee ottaa huomioon millaisia yrityksen salassa pidettävät tiedot ovat.

Tietoturvaskannerien lisäksi testaajien tulisi arvioida järjestelmän tietoturvaa testaajan toimesta. Yksi tärkeä asia, johon tulisi kiinnittää huomiota on se, millä tavalla kadonneet käyttäjätunnukset ja salasanat palautetaan käyttäjälle.

3.4.2 Suorituskykytestaus

Suorituskykytestauksella tarkoitetaan sovelluksen suorituskyvyn testaamista. Suorituskykytestaus voidaan jaotella moneen eri osaan. Sillä voidaan tarkoittaa sitä, kuinka nopeasti sivut latautuvat tai kuinka nopeasti järjestelmä vastaa pyyntöihin, tai esimerkiksi sitä, miten sovellus selviää useasta samanaikaisesta palvelupyynnöstä ja miten palvelupyyntöjen määrä vaikuttaa sovelluksen suorituskykyyn. (Sacks, 2012)

Stress testing on yksi suorituskykytestauksen muoto. Sillä etsitään järjestelmän ”kipupisteitä” eli esimerkiksi samanaikaisten käyttäjien ja/tai palvelupyyntöjen maksimia. Tällä testaustavalla voidaan myös kiinnittää huomiota saatavuuteen. Samalla voidaan testata, miten järjestelmä reagoi maksimikapasiteetti täyttymiseen, eli miten järjestelmä selviytyy ääritapauksista. (Di Lucca *et al., 2006)*

Suorituskykytestauksella löydetään ensisijaisesti ongelmia ajonaikaisesta ympäristöstä. Kuitenkin suorituskykytestauksella voidaan myös tunnistaa, että toteutusratkaisut ovat tehottomia ja tietyn toiminnallisuuden toteutusta pitäisi harkita uudelleen. (Di Lucca *et al*., 2006)

Joskus on myös tarpeellista testata muuttuuko sovelluksen suorituskyky tai ilmeneekö siinä muita ongelmia sen jälkeen, kun sovellus on ollut pitkään ajossa. Tällaisella testauksella pystytään esimerkiksi saamaan kiinni muistivuotoja. Tällaista testausta kutsutaan nimellä *sustained load testing*. (Sacks, 2012)

3.4.3 Saavutettavuustestaus

Saavutettavuustestauksella tarkoitetaan sovelluksen testaamista niissä olosuhteissa, kun sovellusta käyttää henkilö, jolla on erityistarpeita. Käyttäjällä voi olla esimeriksi kuulovamma, näkövamma tai joku muu sovelluksen käyttöä rajoittava vamma. Näillä käyttäjillä on yleensä apusovelluksia tai apuvälineitä sovellusten käyttöön ja testatessa tulisikin varmistaa, että web-sovellus tukee myös näiden käyttäjien tarpeita. W3C-organisaatio ylläpitää *Web Content Accessibility Guidelines* –ohjeistusta, jolla ohjataan saavutettavuuden toteuttamista ja testaamista web-sovelluksissa (W3C, 2008.) Ohjeistus antaa myös muutenkin hyviä ohjeita web-sovellusten toteuttamiseen.

Web-sovellukset, joissa on yleensä paljon JavaScript-koodia, ja jotka nojautuvat paljon visuaaliseen toteutukseen eivät ole saavutettavia sivustoja. Tällaisella toteutuksella äänellä ohjattavat selaimet ja puhesynteesimenetelmät eivät välttämättä toimi. (Wilde, 2007)

3.4.4 Käytettävyystestaus

Käytettävyys on tärkeä laatuominaisuus web-sovelluksissa. Ohjelmistojen käytettävyys on oma tieteenalansa, ja olisikin hyvä, että web-sovellusten käytettävyyden arvioisi alaa opiskellut käytettävyysasiantuntija. Käytettävyys tulisi ottaa huomioon jo sovelluksen suunnitteluvaiheessa. Myös testaajien tulee testatessaan kiinnittää huomiota järjestelmän käytettävyyteen ja raportoida löytämistään ongelmista.

Jos yrityksessä ei ole käytettävyysasiantuntijaa, tulisi testaajia kouluttaa käytettävyyteen ja sen testaamiseen. Esimerkiksi heuristista evaluointia voidaan käyttää apuna. Vaikka sen tuloksia suositellaan asiantuntijoiden tulkittavaksi, on myös todettu, että muut kuin alan ekspertit voivat saada myös hyviä tuloksia. (UsabilityNet, 2015)

3.4.5 Yhteensopivuustestaus

Yhteensopivuustestauksella tutkitaan, miten web-sovellus toimii erilaisissa suoritusympäristöissä. Perinteisissä sovelluksilla pitää tyypillisesti ottaa huomioon eri käyttöjärjestelmät ja tietokoneiden suorituskykyjen erilaisuudet. Ajatellaan, että web-sovellusten on tarpeellista toimia lukuisilla eri alustoilla. Erilaisia alustoja ovat sekä käyttöjärjestelmät että selaimet. Usein selaimistakin on lukuisia eri versioita. (Software Testing Help,

2015/2) Erityisesti Internet Explorer selaimesta käytetään useita eri versioita, koska esimerkiksi vanhaan Windows XP järjestelmään ei saa asennettua Internet Explorerista uudempaa versiota kuin version kahdeksan.

Testausta voidaan suorittaa käyttämällä web-sovellusta eri selaimilla, eri alustoilla ja erilaisilla asetuksilla näissä ympäristöissä. (Di Lucca *et al*., 2006). Tässä testausmuodossa on tärkeää tietää, millaisia käyttöympäristöjä loppukäyttäjillä on. On esimerkiksi tärkeää tietää, käytetäänkö sovellusta mobiililaitteilla ja mitä versioita tietystä selaimesta on käytössä järjestelmässä. Tämän tyyppistä testausta varten on tarpeellista kartoittaa kaikki mahdolliset käyttötapaukset sekä -ympäristöt ja tiedostaa, mitä merkitystä järjestelmälle on, jos sen käyttöä ei tueta jollain alustalla tai suoritusympäristössä.

3.5 Testiautomaatio

Testausautomaatiota voidaan suorittaa kaikilla testauksen tasoilla, eli sekä toiminnallisella että ei-toiminnallisella puolella. Erilaisia työkaluja testien automatisointiin on tarjolla paljon. Näistä osa on *open source* -ohjelmia ja osa taas hyvinkin kalliita kaupallisia ohjelmistoja.

Tarve automaatiotestaukselle kasvaa koko ajan järjestelmien kasvaessa. Automaatiotestauksen parantaminen onkin yksi tulevaisuuden haaste web-sovellusten testaamisessa. Yksi kehityssuunta olisi esimerkiksi yhä parempi tuki testien automaattiselle luomiselle esimerkiksi käyttäjien käyntien perusteella. (Di Lucca *et al.*, 2006)

Automaattisissa testeissä tulee huomioida niiden ylläpidon tarve. Monet organisaatiot saattavat päätyä hylkäämään testiautomaation hyödyntämisen jo muutaman kuukauden jälkeen. He kokevat, että testien ylläpidosta on liikaa työtä. Järjestelmän muuttuessa myös automaatiotestit tulee päivittää, jotta ne toimivat. (Emery, 2009) Web-sovellusten testaamisessa helpottaa, jos sama automaatiotesti on ajettavissa kaikilla selaimilla, eikä joka selaimelle tarvitse ylläpitää omaa testiä. Tällöin järjestelmän muutosten myötä tarvitsee päivittää vain yhtä testitapausta.

Testiautomaatiotyökaluilla voidaan tehdä käyttöliittymän kautta tapahtuvaa testausta. Tällä tavalla pystytään luomaan järjestelmätestaukseen käyttöliittymän kautta tapahtuvaa regressiotestausta, eli jatkuvaa testausta, joka tapahtuu muutosten yhteydessä. Tämä on kuitenkin etenkin monimutkaisissa järjestelmissä erittäin aikaa vievää ja tehotonta. (Benedikt *et al.*, 2002) Automaatiotestejä pidetään järkevinä, koska web-sovellusten testaaminen on sovellusten monimutkaisuuden takia manuaalisesti tehtynä hyvin kallista.

Automaatiotestausta ei kannata aloittaa vain koska se on trendikästä, vaan sen tarvetta tulee harkita tarkkaan. Automaatiotestejä harkittaessa pitää vastata moniin eri kysymyksiin: Mihin automaatiotestejä halutaan? Millainen yrityksen testausprosessi on, ja mihin erityisesti kuluu aikaa? Tehdäänkö jotain testausta manuaalisesti, ja mikä sen tarkoitus

on? Kehittäessään testausautomaatiota yrityksen pitää tunnistaa testaustarpeensa tarkkaan. Testiautomaation käyttöönotto voi olla kallis prosessi ja jos työkalun evaluoinnissa on epäonnistuttu, yritys voi heittää hukkaan sekä ajallisia että rahallisia resursseja.

3.6 Web-sovellusten testauksen erikoispiirteet

Seuraavaksi käydään läpi joitakin web-sovellusten piirteitä, joiden takia näiden sovellusten testaaminen on haastavampaa kuin perinteisten sovellusten.

Web-sovellusten kehitysprosessissa korostuu tarve toimittaa tuote nopeasti asiakkaalle. Tästä syystä web-sovellusten testaukselle saattaa jäädä vielä vähemmän aikaa kun perinteisten sovellusten testaamiselle. Kehitysprosessissa korostuu nykykehitysmetodologioilla se, ettei web-sovelluksen määrittely ole alussa lyöty lukkoon, vaan muutoksia saatetaan tehdä hyvinkin myöhäisessä vaiheessa kehitysprosessia. Tämän takia web-sovellusten testaamiseen tarvitaan automatisoitua ja joustavaa testausta. (Dobolyi, 2010)

Web-sovellukset ovat muuttuneet monimutkaisiksi kokonaisuuksiksi, jolloin myös sovellusten testaustarve korostuu kun virheiden mahdollisuus on suurempi. Web-sovellusten dynaaminen luonne vaikeuttaa sovellusten testaamista. Käyttäjien suuren määrän takia on mahdollista, että käyttäjät luovat sellaisia toimintaketjuja, joita testaajat eivät osanneet ennustaa, ja saavat sovelluksen joillain tavalla toimimaan ei-toivotusti. (Benedikt *et al*., 2002)

Web-sovelluksen testauksessa korostuu virheiden jaottelu kahteen sen mukaan, aiheutuvatko ne ohjelman koodillisista ongelmista vai ympäristöstä johtuvista syistä. Tämä erottelu ei ole aina selvää ja testaamiseen tarvitaankin useampia testausympäristöjä. Web-sovelluksen toiminnalliseen puoleen vaikuttavat enemmän kehittäjien tekemät virheet ja testausympäristö taas vaikuttaa enemmän ei-toiminnalliseen puoleen, kuten suorituskykyyn. (Di Lucca *et al*., 2006)

Dynaamisissa web-sovelluksissa voi tulla ongelmaksi testien uudelleen tuottaminen, sillä dynaamisuutensa takia web-sovellusten sivun sisältö voi vaihdella suurestikin käyttäjän syötteistä riippuen (Xu *et al.,* 2004). Saman tilanteen luominen ja vikojen toistaminen voikin olla vaikeaa juuri web-sovellusten dynaamisen luonteen takia (Di Lucca *et al*., 2006).

3.7 Olemassa olevan web-sovelluksen testaaminen

Edellisissä kohdissa on käyty läpi web-sovellusten testaamisen pääpiirteet ja eroteltu eri testaamisen osa-alueet. Kuitenkin testausstrategian luominen valmiiseen web-sovellukseen vaatii hieman erilaista lähestymistapaa kuin testausstrategian luonti sovelluksen kehitysvaiheessa.

Pitkään kehityksessä olleet sovellukset, joita ei ole dokumentoitu kunnolla, ovat helposti kasvaneet hallitsemattomiksi. Samaan ominaisuuteen vaikuttavaa toiminnallisuutta voi olla hyvin monessa eri paikkaa koodia ja muutokset johonkin osaan koodia saattavat vaikuttaa odottamattomalla tavalla johonkin muuhun ominaisuuteen. Tämä näkyy erityisesti web-sovelluksissa, joissa ohjelman rakenne ei välttämättä ole yhtä selkeä kuin perinteisissä sovelluksissa. Erilaisten riippuvuuksien takia on vaikea rajata ajettavaa testijoukkoa, ja testatessa pitää käyttää aikaa myös uusien tai muutettujen toiminnallisuuksiin liittymättömien testien ajamiseen. (Rice, 2012)

Olemassa olevan web-sovelluksen testaamisen avainsanana on automaattinen regressiotestaus järjestelmätasolla. Sillä tavalla pystytään varmentamaan, että käyttäjä pystyy edelleen käyttämään järjestelmää ja tärkeimmät toiminnallisuudet eivät ole hajonneet muutosten yhteydessä. (Rice, 2012; Kessler, 2012)

4. TESTAUSTRATEGIAN LUOMINEN WEB-SOVELLUKSEEN

Tässä luvussa esitellään kohdejärjestelmänä oleva web-sovellus sekä sille luotu testausstrategia. Testausstrategian eri osat käydään läpi eri toiminnallisen testauksen tasojen ja ei-toiminnallisen testauksen osa-alueiden kautta. Luvun lopussa on lyhyt yhteenveto testausstrategiasta taulukkomuodossa.

4.1 Kohdejärjestelmä

Tässä kohdassa esitellään aluksi kohdejärjestelmä yleisesti. Sen jälkeen esitellään järjestelmän toteutusteknologioita sekä käyttäjiä. Kohdan tavoite on luoda lukijalle kuva kohdejärjestelmästä ja sen erityispiirteistä, jotka ovat vaikuttaneet testausstrategian suunnitteluvaiheessa.

4.1.1 Kohdejärjestelmä yleisesti

Kohdejärjestelmä on web-sovellus, joka on riskienhallinnan työkalu yrityksille ja organisaatioille. Järjestelmän tarkoitus on helpottaa päivittäistä riskienhallintaa asiakasyrityksissä. Työkalulla voidaan esimerkiksi laatia ilmoituksia tapahtuneista vaaratilanteista sekä arvioida riskien todennäköisyyksiä ja vakavuuksia. Työkalulla voidaan toteuttaa myös erilaisia kartoituskyselyitä. Lisäksi järjestelmän kautta voidaan toteuttaa koulutuksia asiakasyritykselle. Eräät lait ja standardit vaativat tietyntyyppisiltä yrityksiltä ja organisaatioilta vaatimustenhallinnointia riskeihin ja työolosuhteisiin nähden, ja tämä onnistuu kohdejärjestelmällä.

Järjestelmässä on viisi eri moduulityyppiä. Näistä moduuleista luodaan toteutuksia, johon voidaan asettaa käyttäjiä. Peruskäyttäjät näkevät toteutukset, joille heidät on asetettu. He voivat moduulityypistä riippuen tehdä eri toimintoja, kuten suorittaa tietoturvakoulutuksia tai tehdä vaaratilanneilmoituksia. Asiakkaalla voi olla erikseen peruskäyttäjiä, jotka suorittavat vain toteutuksia sekä hallintakäyttäjiä, jotka pystyvät tarkastelemaan toteutuksien tuloksia. Asiakkaan hallintakäyttäjille voidaan antaa sisällöntuottajaoikeudet, jolloin hän pystyy muokkaamaan moduulia ja sen sisältöä. Pääkäyttäjän oikeuksilla kaikki järjestelmän ominaisuudet ovat käytettävissä. Eri käyttäjätyyppien lisäksi käyttäjille voidaan antaa erilaisia yhteenvetonäkymäoikeuksia, joilla he pystyvät yhteenveto-näkymässä tarkastelemaan toteutuksien etenemistä ja tuloksia erilaisten raporttien avulla.

Jokaiselle asiakkaalle voidaan räätälöidä oma variantti kohdejärjestelmästä asiakkaan tarpeiden mukaan. Palvelua tarjotaan sekä asiakkaan omalle palvelimelle asennettavaksi tai

tarjoavan yrityksen palvelimella pyöriväksi. Asiakkaat ovat valinneet molempia ratkaisuja.

Järjestelmää on kehitetty vuodesta 2005 ja siitä on julkaistu 30 – 35 versiota. Versionumerona lokakuun 2014 versiossa on 4.10. Vuosina 2013 ja 2014 versiojulkaisuja on tehty vuoden aikana kolme, ja myös vuodelle 2015 on suunniteltu kolme uutta versiota järjestelmästä. Uusien versioiden muutokset ovat sekä vikakorjauksia että uusien toiminnallisuuksien lisäämistä järjestelmään. Tuotteen toimittaminen asiakkaille on jo vakiintunut prosessi, joka sujuu hyvin.

Järjestelmän toimintaa mobiililaitteilla ei ole tuettu järjestelmän vanhemmissa versioissa, mutta vuoden 2015 maaliskuun versioon käyttöliittymäuudistuksen myötä mobiilituki tuotiin järjestelmään. Käyttöliittymäuudistuksella ei haluttu pelkästään tuoda tukea mobiililaitteille, vaan myös päivittää vanhentunutta ulkonäköä ja parantaa toiminnallisuuksia.

Kohdejärjestelmää myydään vain yrityskäyttöön. Asiakkaina on yrityksiä, teollisuuslaitoksia, pankki- ja finanssialan yrityksiä sekä kuntia, terveydenhuollon organisaatioita ja valtionhallinnon organisaatioita. Koska asiakasorganisaatioita on monenlaisia, myös loppukäyttäjien tietotaso ja atk-käyttötaidot vaihtelevat suuresti.

Pääasiallinen järjestelmän käyttökieli on suomi. Järjestelmää tarjotaan myös muilla kielillä kuten ruotsiksi ja englanniksi. Muille kieliversioille järjestelmässä on tuki ja asiakkaiden tarpeiden mukaan järjestelmästä voidaan tarjota eri kieliversioita. Lisäksi osa suomalaisista asiakkaista käyttää järjestelmää ruotsiksi. Tarkkaa tilastoa käytetyimmistä selaimista ei ole, mutta oletettavasti eri Internet Explorerin versiot ovat käytetyimmät asiakaspään selaimet. Tukipyyntöjä tulee eniten näiden selainten käyttäjiltä, mutta tämä ei välttämättä suoraan kerro käytetyistä selaimista.

Pääasiallisesti asiakkaat käyttävät kohdejärjestelmää päivisin, mutta järjestelmällä on myös jonkin verran vuorotöistä johtuvaa ympärivuorokautista käyttöä. Asiakkaille tarjotaan käyttötukea arkipäivinä normaaleina työaikoina. Tuessa asiakaspyynnöt käsitellään parin tunnin vasteajalla.

Käyttäjiltä itseltään saadaan järjestelmästä paljon palautetta, niin kirjallisesti sähköposteilla kuin suullisesti. Asiakkaille järjestetään kaksi kertaa vuodessa asiakasseminaari, jossa otetaan vastaan palautetta ja informoidaan tulevista muutoksista. Asiakkaisiin pidetään myös yhteyttä asiakasseminaarien välissä, ja heiltä kysellään mielipiteitä palvelusta. Palautteen avulla voidaan järjestelmän kehityksessä ottaa huomioon asiakkaiden toiveet.

4.1.2 Toteutusteknologiat

Kohdejärjestelmä on perinteisillä web-toteutusteknologioilla toteutettu web-sovellus. Toteutusteknologioita ovat palvelinpäässä PHP Doctrine- ja Smarty-kirjastoilla, ja selainpäässä toteutusteknologioina ovat HTML5, CSS sekä JavaScript jQuery-kirjastolla.

Palvelinpäässä PHP valittiin toteutuskieleksi, kun järjestelmän kehitys aloitettiin vuonna 2005. Tällöin PHP koettiin järkeväksi valinnaksi, koska järjestelmän kehittäjillä oli siitä kokemusta ja silloiset toteutusteknologiat eivät tarjonneet hyvää vaihtoehtoa. Yritys piti PHP:ssa hyvänä sen nopeutta muutosten tekemiseen ja näiden muutosten vaikutuksen nopean tarkastamisen mahdollisuutta. Valittaessa kieltä PHP:ta arvostettiin myös laajan käyttöjärjestelmätuen takia, joka ei aseta rajoituksia asennusympäristölle. PHP:n huonoimmaksi puoleksi yrityksessä koetaan säännönmukaisuuksien puute. Tällä tarkoitetaan, että samantyyppisiä asioita toteutetaan hieman eri tavalla eri paikoissa. Yrityksen teknologiapäällikkö kommentoi, että jos toteutuskielen valinta tehtäisiin nyt, ei PHP nousisi kovin korkealle toteutusteknologian ehdokaslistassa. Teknologiapäällikkö kuitenkin toteaa, että PHP on tullut tutuksi vuosien aikana vanhemmille työntekijöille ja sen kanssa on pärjätty hyvin.

Järjestelmän JavaScript-kirjastoksi on valittu jQuery. JQueryn vahvuuksia on sen pienuus, yksinkertainen syntaksi, hyvä dokumentaatio sekä suuri online-yhteisö. Lisäksi esimerkiksi metodien ketjuttaminen on jQueryn etuna. (Lengstorf, 2010)

Arkkitehtuurimallina kohdejärjestelmässä on MVC-arkkitehtuuri. Järjestelmään on suunniteltu arkkitehtuuriuudistusta, joka jakaisi järjestelmän pienempiin osiin ja mahdollistaisi rajapintojen hyödyntämisen. Tällöin myös toteutusteknologioiden päivittäminen voisi tulevaisuudessa olla mahdollisesti helpompaa. Versionhallinnan alaista koodia on noin 800 000 koodiriviä, joista kuitenkin huomattava osa on avoimen lähdekoodin kirjastoja.

4.1.3 Yrityksen prosessit ja työkalut

Projektinhallinnassa yrityksellä on käytössä kevennetty Kanban. Kanban on agile-menetelmä (Scotland, 2010). Yrityksellä on tehtävienhallintaohjelmisto JIRAssa vika- ja kehitystiketit, joille on määritelty tärkeysaste (JIRA, 2015). Teknologiapäällikkö yhdessä toimitusjohtajan kanssa määrittää, mitä seuraavaan versiojulkaisuun halutaan toteutettavaksi ja asettaa JIRAssa nämä tiketit version tehtävälistaksi. Tästä tehtävälistasta kehittäjät ottavat itselleen tehtäväksi tikettejä teknologiapäällikön suosittelujen mukaisesti.

Kerran viikossa kehittäjät pitävät palaverin, yleensä Skypen välityksellä (Skype, 2015). Tällöin käydään läpi asiakastiketit ja keskustellaan asiakastapauksista. Palaverissa käydään myös lyhyesti läpi mitä kukin työstää sillä hetkellä ja miltä seuraavan version aikataulu näyttää.

Kun järjestelmästä löydetään jokin ongelma testauksen tai muun kehitystyön yhteydessä tai asiakkaan ilmoittamana, tehdään siitä JIRAan vikatiketti. Teknologiapäällikkö käsittelee kaikki vikatiketit ja määrittelee mihin versioon ne korjataan, vai tehdäänkö niistä pikakorjausversio.

Jokaisella kehittäjällä on oma lokaali ympäristönsä, jossa he tekevät kehitystyötä. Kun he ovat saaneet valmiiksi jonkin ominaisuuden tai virhekorjauksen, he versionhallinnan avulla lisäävät sen kehitysversioon, jonka muut saavat haettua versionhallinnan avulla. Beta-ympäristössä on käytössä kehitysversio. Beta-ympäristön päivitykseen on tarjolla erillinen sivu, josta saa päivitettyä ympäristön version vastaamaan viimeisintä versiota, joka on versionhallinnassa kehitysversiona. Versiojulkaisua varten luodaan oma ”stable”-versio, joka päivitetään Gpdemo-ympäristöön ja sen jälkeen myös asiakkaille. Tähän versioon voidaan tehdä pikakorjauksia tarvittaessa.

Yritys haluaa, että asiakkaiden vikailmoituksiin vastataan nopeasti. Tämä tarkoittaa, että pikakorjauksia julkaistaan tarpeen vaatiessa, jos järjestelmästä löytyy isompia ongelmia. Pienemmät vikakorjaukset sisällytetään kuitenkin versiojulkaisuihin. Yksi työntekijöistä hoitaa asiakastukea, ja asiakastuessa on taattu 2-3h vastausaika asiakkaille.

JIRAn kanssa samasta tuoteperheestä yrityksellä on käytössä Confluence (Confluence, 2015). Confluencessa hallitaan yrityksen liiketoimintatietoja ja ylläpidetään dokumentaatiota asiakkaista ja heidän ympäristöistään. Lisäksi Confluencessa on dokumentaatiota yrityksen prosesseista, mutta ne saattavat olla vanhentuneita joiltain osin.

4.2 Testausstrategian luomisen lähtötilanne

Yrityksellä on yhteensä 9 työntekijää. Näistä työntekijöistä kaksi on kokoaikaisia kehittäjiä, yksi osa-aikainen kehittäjä, yksi osa-aikainen testaaja, yksi projektiassistentti sekä yksi projektipäällikkö, joka hoitaa pääasiassa myyntiä. Näiden lisäksi yrityksessä on 3 myyjää. Osa myyjistä on kokoaikaisia ja osa osa-aikaisia. Myyjistä projektipäällikkö voi osallistua tuotteen testaamiseen tarvittaessa. Kaikki kehittäjät voivat osallistua testaamiseen rajoitetusti uuden version valmistumisen jälkeen. Kehittäjillä on vastuu testauksen aikana löytyneiden virheiden korjaamisesta.

Työntekijöistä kehittäjät, projektipäällikkö ja -assistentti vastaavat tukipäivystyksestä niin, että jokaisena päivänä joku tukivuorolaisista on päivystysvuorossa. Tähän kuuluu, että tukipyyntöjä käsitellään päivän aikana niin, ettei asiakkaalle vastaamisaika pitene yli neljään tuntiin. Myös testaaja osallistuu tukipäivystyksiin helmikuusta 2015 lähtien.

Kohdejärjestelmää on tarjolla eri kehitysympäristöissä. Näistä Beta-ympäristö on tarkoitettu puhtaasti testaamista varten, ja tähän ympäristöön asiakkailla ei ole pääsyä. Beta-ympäristöön saatetaan päivittää muutoksia useita kertoja päivässä, koska se päivitetään kun kehittäjä on saanut jonkin vika- tai kehitystiketin valmiiksi. Gpdemo-ympäristö on

ympäristö, jonka avulla uusille ja mahdollisille asiakkaille esitellään järjestelmää. Lisäksi Gpdemo-ympäristöä käytetään uuden version esittelyyn vanhoille asiakkaille. Asiakkaat voivat käydä halutessaan testaamassa uutta versiota Gpdemo-ympäristössä ennen kuin se asennetaan asiakkaalle käyttöön.

4.2.1 Testaus ennen testausstrategiaa luomista

Kohdejärjestelmää ovat pääasiallisesti testanneet kehittäjät toteuttaessaan uusia ominaisuuksia. Tämän testauksen lisäksi tehtiin järjestelmätestausta versiojulkaisun yhteydessä. Järjestelmätestauksessa suoritettiin manuaalitestejä, joilla varmistettiin, että tärkeimmät ominaisuudet eivät ole menneet rikki. Version julkaisusta päätti teknologiapäällikkö. Teknologiapäällikkö antoi julkaisuluvan kun ennalta määritellyt manuaalitestit oli ajettu järjestelmään ja ne suoritettiin onnistuneesti. Tämä on toiminut hyväksymistestauksen tavoin.

Yrityksellä on ollut muutamia osa-aikaisia testaajia, kuitenkin vain yksi henkilö kerrallaan. Testaajien vaihtuvuus on ollut suurta. Testaajat ovat olleet niin lyhytaikaisia työntekijöitä, että he eivät ole ehtineet ottamaan järjestelmää kunnolla haltuun vaan testaus on jäänyt oletettavasti pintapuoliseksi eikä testausprosessia ei ole saatu kehitettyä. Testaajien vaihtuvuus on hidastanut kunnon testausstrategian syntyä sekä testauskäytäntöjen vakiintumista. Yrityksen edustaja sanoo, että pahimmillaan laadunvarmistuksessa on epäonnistuttu niin, että sovelluksen loppukäyttäjät ovat antaneet palautetta, jonka mukaan he ovat kokeneet joutuneensa järjestelmän testaajiksi. Yleisesti järjestelemän laatua asiakkaat pitävät yrityksen arvion mukaan kuitenkin kohtalaisesta hyvänä.

Yksikkötestejä ei otettu alun perin käyttöön, koska yrityksessä koettiin yksikkötestauksen olevan hankalaa ja työlästä. Yksikkötesteistä yrityksen edustajat totesivat, ettei asiaan ole varmaankaan perehdytty tarpeeksi. Yrityksessä myös arveltiin, ettei PHP:n tuki yksikkötestaukselle varsinkaan järjestelmän kehittämisen alkuvaiheessa ollut riittävän hyvä.

Yritys oli yrittänyt aiemmin luoda Selenium-testiautomaatiokehyksellä testiautomaatiojärjestelmää. Selenium todettiin sopimattomaksi, koska se ei tukenut riittävästi eri selaimilla tapahtuvaa testausta. Seleniumilla tarvittiin käyttöön erilaisia odotusaikoja skripteihin, ja ne piti tehdä selainkohtaisesti. Tämä koettiin raskaaksi ja huonosti ylläpidettäväksi systeemiksi. Selenium-työkalua yhdistettiin Hudsoniin ja tällä yhdistelmällä kokeiltiin jatkuvaa integraatio -menetelmää. Tämä ei kuitenkaan toiminut, kun Selenium todettiin sopimattomaksi yrityksen tarpeisiin.

Tietoturvatestaukseen yrityksessä on käytetty Acunetix-web-tietoturvaskanneria. Erillistä suorituskykytestausta ei ole tehty, vaan asiakkaiden palautteiden kannalta on arvioitu sivulatausten hitautta. Lisäksi sivulatauksia on tarkkailtu muun testauksen yhteydessä.

Myöskään saavutettavuustestausta ei järjestelmälle ole tehty, eikä saavutettavuusnäkökulmaa ole otettu huomioon järjestelmää suunniteltaessa. Toistaiseksi järjestelmän elinaikana ei ole tullut tarvetta erityisille saavutettavuusratkaisuille.

Testauksessa ei ole hyödynnetty sitä, että kohdejärjestelmä kerää dataa sivulatauksista. Lokeista selviää käyttäjä, sivu ja sivun latausaika. Tätä tietoa voitaisiin käyttää esimerkiksi suorituskykytestaukseen ja käyttäjien käyttäytymisen perusteella luotaviin testitapauksiin.

4.2.2 Yrityksen tavoitteet testausstrategialle

Yritys halusi lähteä kehittämään testausta, jotta yrityksen tuotteesta saataisiin entistä laadukkaampi, ja samalla myynti ja markkinointi helpottuisivat. Tuotekehitykseen on alettu panostaa enemmän yrityksen myynnin lisääntyessä. Yrityksen resurssien kasvaessa testaukseen on haluttu panostaa enemmän. Joulukuussa 2014 yritys arvioi, että noin puolta työviikkoa tekevä testaaja vastaisi yrityksen tarpeita. Yritys pyrkii kasvamaan ja lisäämään myyntiään niin, että vuoden sisällä pystyttäisiin jo palkkaamaan kokoaikainen testaaja. Testaajan vastuualueisiin kuuluu testiautomaatioiden luominen ja ylläpitäminen sekä järjestelmän testaus testausstrategian mukaisesti. Lisäksi testaajalle kuuluu testauksesta raportoiminen sekä järjestelmä- että hyväksymistestausvaiheessa testauksen organisointi.

Testausautomaatiota yritys haluaa lisätä. Testausautomaatiolla halutaan saada katettua peruskäyttötapaukset, jotka on ennen testattu manuaalisesti. Manuaalitestien suorittaminen on vienyt paljon aikaa myös kehittäjiltä versiotestauksen yhteydessä. Tämä vie aikaa muulta kehitystyöltä.

Automaatiotestien tulisi olla helposti ajettavissa, jotta niillä pystytään päivittäin tai kerran viikossa tarkastamaan, ovatko järjestelmän uudet implementaatiot aiheuttaneet vikaa vanhoissa ominaisuuksissa eli testiautomaatiolla halutaan suorittaa järjestelmätestauksen regressiotestausta. Testiautomaation pitäisi olla testaajan ja kehittäjän helposti suoritettavissa ja tulosten luettavissa. Automaatiotestien tulisi kattaa selaimista ainakin FireFox, Internet Explorer ja Chrome, ja mielellään myös Safari. Testiautomaatiotyökalua ei ole valittu yrityksen puolesta, vaan testausstrategian toteuttajalle annetaan vastuu työkalun valitsemisesta.

4.3 Testausstrategia

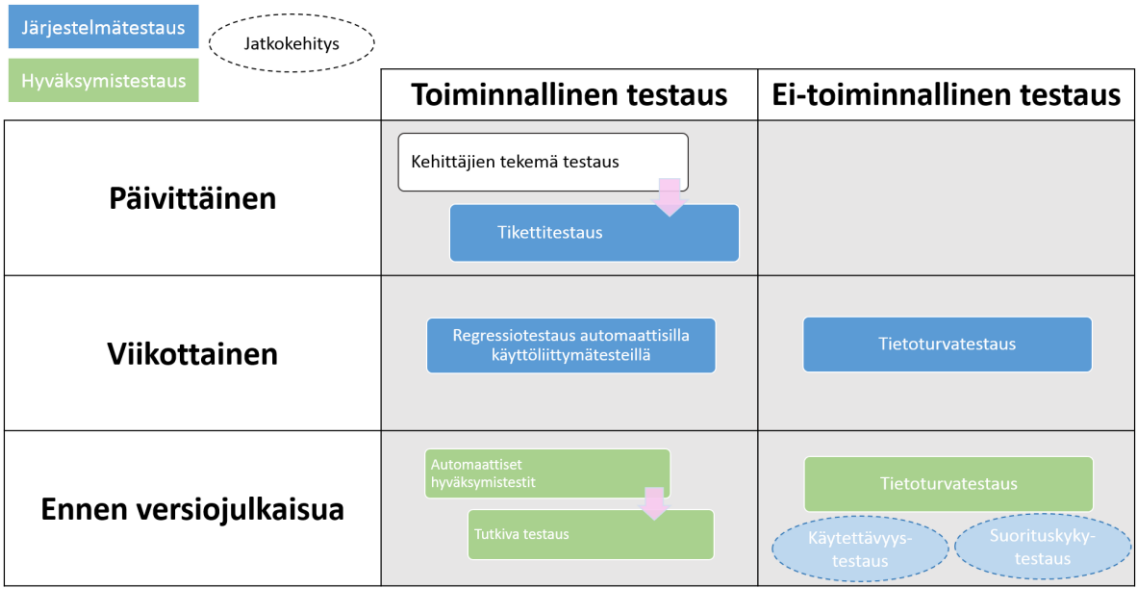
Testausstrategiaa aloitettiin luomaan yrityksen asettamien lähtökohtien pohjalta. Testausstrategian luomisen alussa todettiin, että haluttu pääasiallinen testaustapa on käyttöliittymän kautta tapahtuva testaus. Tämän lähestymistavan tarpeellisuus todettiin kohdassa 3.7. Käyttöliittymätestauksen avulla voidaan arvioida, millainen kokemus käyttä-

jille tulee järjestelmää käytettäessä. Lisäksi käyttöliittymätestauksessa pystytään varmistamaan, että käyttöliittymä toimii oikein ja näyttää oikealta. Tällä tavalla pystytään puuttumaan myös järjestelmän käytettävyysongelmiin. Yrityksellä ei ole erityistä käytettävyysosaajaa, joten testaajan tulee ottaa kantaa myös järjestelmän käytettävyyteen.

Web-sovellusten teoriaan tutustumisen jälkeen, yrityksen entistä testausprosessia analysoimalla ja yrityksen tavoitteita hyödyntämällä testausstrategian tärkeimmiksi kohdiksi valittiin seuraavat:

1) Automatisoitu regressiotestaus järjestelmätesaustasolla käyttöliittymän kautta
2) Tutkiva testaus uusien ominaisuuksien toimivuuden arvioimiseksi
3) Tietoturvatestauksen ylläpitäminen

Kuvassa 3 on esitetty testausstrategiaan suunnitellut vaiheet. Kuvassa väreillä on kuvattu millä testaustasolla testaaminen tapahtuu. Kuvan jälkeen ensimmäisessä alakohdassa on esitelty mitä muutoksia yrityksen tietojärjestelmiin tehtiin testausstrategian käyttöönottoa varten. Seuraavissa alakohdissa on esitelty ensin testausstrategian muodostuminen eri testaustasojen kautta. Toiminnallisen testauksen jälkeen alakohdissa on käyty läpi ei-toiminnalliset testaukset samalla tavalla jaoteltuna kuin aiemmin. Osa alakohdista perustelee miksi mikään vaihe ei toteuta kyseistä testaustasoa ja osassa alakohdista on kuvattu tason testausvaihe.



Kuva 3. *Testausstrategian mukaiset testausvaiheet*

Testausstrategiaan kuuluu tärkeänä osana testausstrategian kehittäminen jokaisen versiojulkaisun jälkeen. Testausstrategian ei oleteta olevan ensimmäisellä kerralla täydellinen, vaan sen kehittäminen vie useamman versiojulkaisun.

4.3.1 Muutokset tietojärjestelmiin

Uutta testausstrategiaa käyttöönotettaessa tehtiin muutoksia myös yrityksen tietojärjestelmiin. Yrityksen käyttämään JIRA-tehtävienhallintaohjelmistoon lisättiin workflow'n In testing –tila, johon kehittäjät siirtävät testausta vaativat tiketit. Workflow sisältää nyt *Open, In progress, In testing, Closed* ja *Resolved* -tilat. Tällä tavalla pystytään seuraamaan, moniko tiketti odottaa tarkempaa testausta. Päätiketteihin lisättiin myös Tester(testaaja)-kenttä. Tätä kenttää käyttämällä voidaan määrittää, kuka yrityksen työntekijöistä testaa tiketin. Jokaisella työntekijällä tulisi olla omassa aloitusnäkymässään JIRAssa suodatettuna ne tiketit, joihin hänet on asetettu Tester-kenttään.

Lisäksi testauksen seurantaa varten luotiin alatikettityyppi Testing task (testaustehtävä). Jokaiseen testattavaan tikettiin luodaan testing task -tyyppinen alatiketti, johon testaajat kirjaavat ylös, paljonko he käyttivät aikaa kyseisen päätiketin testaamiseen. Tällä tavalla pystytään seuramaan tarkemmin testaukseen kulunutta aikaa ja arvioimaan testaustarvetta jatkossa paremmin.

4.3.2 Yksikkötestaus

Lähtötilanteessa web-sovelluksessa on satoja tuhansia koodirivejä eikä yhtäkään yksikkötestiä ole kirjoitettu. Yksikkötestien tuomista projektiin uuden testausstrategian myötä harkittiin, mutta todettiin, ettei se ole tässä vaiheessa järkevää. Julkaisuaikataulut ovat tiukat, ja nytkin kehityksessä ollaan koko ajan hieman jäljessä aikataulusta. Lisäksi kehittäjät ovat pohtineet järjestelmän rakenteen uudistamisen tarvetta. Yksikkötestien mukaan tuomista pitäisi harkita järjestelmän uudistamisvaiheessa tarkemmin.

4.3.3 Integraatiotestaus

Yksikkötestien puutteen takia järjestelmässä ei myöskään ole erillistä integraatiotestausta, vaan kehittäjät vastaavat koodin toiminnallisuuden testaamisesta. Selvä haitta tästä on, ettei järjestelmästä saada minkäänlaista kattavuusdataa. Suuri osa järjestelmästä voi jäädä kokonaan testauksen ulkopuolelle ja pahojakin ongelmia voi päästä kehittymään. Toisaalta järjestelmässä voi olla myös koodia, jota ei ole tarpeen kattaa testeillä koska se on vanhentunutta eikä se vaikuta sovelluksen toimintaan enää.

4.3.4 Järjestelmätestaus

Järjestelmätestaus päätettiin ottaa pääasialliseksi testaustavaksi tälle web-sovellukselle. Järjestelmätestaustasolla voidaan myös toteuttaa kätevästi eri selaimilla järjestelmän testaus, joka listattiin testausstrategiaa tehdessä yhdeksi tärkeimmistä vaatimuksista. Testausstrategiassa useampi vaihe suoritetaan järjestelmätestaustasolla.

Kaikki uudet ominaisuudet ja vikojen korjaukset on esitetty JIRA:n tiketteinä. Kuten aiemmin mainittiin, kehittäjät toiminnallisuuden toteutettuaan siirtävät tiketin *In testing* –tilaan. Tämä tila tarkoittaa, että tiketti on testaajille testattavissa ja toiminnallisuus on päivitetty Beta-ympäristöön. Testaajat valitsevat aina yhden *In testing* -tilassa olevan tiketin itselleen testattavaksi ja merkitsevät tiketin *Resolved*-tilaan eli ratkaistuksi, kun ovat mielestään testanneet tikettiä tarpeeksi. Mikäli tikettiä testatessa ilmenee vikoja, nämä huomiot kirjataan tikettiin. Tällöin kehittäjä saa tiedon virheestä ja ottaa ominaisuuden uudelleen käsiteltäväksi. Tätä testaustapaa kutsutaan tässä työssä **tikettitestaukseksi.**

Tässä lähestymistavassa käytetään testaustapana tutkivaa testausta. Tutkiva testaus vaatii testaajalta ammattitaitoa ja huolellisuutta. Tämä lähestymistapa kuitenkin mahdollistaa helposti eri käyttötapauksien testaamisen tietylle ominaisuudelle. Lisäksi testaaja pystyy muuttamaan testauksen suuntaa tarvittaessa, jos järjestelmä reagoi jollain odottamattomalla tavalla.

Tutkivan testauksen todettiin sopivan erityisesti tämän järjestelmän testaamiseen, koska se antaa mahdollisuuden oppia järjestelmästä ja testata kohdejärjestelmälle tyypillisesti ongelmallisia kohtia. Myös tarkkojen testitapausten kirjoittaminen jokaiselle ominaisuudelle vaikuttaa olevan hidasta, ja tutkivan testauksen kirjallisuus argumentoi tutkivan testauksen tehokkuuden puolesta, kuten luvussa 3.3.3 esitettiin. Järjestelmässä on myös paljon erilaisia suodatus- ja graafiominaisuuksia, joiden oikea tulkinta vaatii testaajan näkemystä.

Tutkiva testaus ei ole myöskään riippuvainen järjestelmän dokumentaatiosta. Hyvä dokumentaatio tukee tutkivaa testauksen suorittamista, mutta se ei perustu dokumentaatiolle niin kuin etukäteen suunnitellut testitapaukset. Hyödyntämällä tätä menetelmää yrityksen testausstrategia saatiin heti käyttöön ja dokumentaation puute ei estänyt testaamisen aloittamista. Tyhjästä valmiiseen järjestelmään dokumentaation luominen on työlästä ja sen järkevää toteuttamista tulee harkita tarkasti.

Tutkivaa testausta käyttämällä voidaan myös hyödyntää yrityksen eri työntekijöiden osaamista. Tutkivalla testauksella kenenkään ei tarvitse dokumentoida ja määrittää tarkasti, mitä testataan, vaan yrityksen työntekijä voi käyttää hiljaista tietoaan hyväksi testauksessa. Esimerkiksi myyjillä on paljon hiljaista tietoa, jota he saavat asiakkaiden kanssa vuorovaikuttaessaan. Testaaja ei ole suorassa yhteydessä asiakkaisiin, joten testaajalla ei ole kertynyt hiljaista tietoa siitä, miten asiakkaat käyttävät sovellusta. Tästä syystä tulisi kehittää tapa, jolla myyjiltä siirrettäisiin tietoa testaajalle ja myyjien osaamista hyödynnettäisiin paremmin.

Erillisen käytettävyysasiantuntijan puuttuessa tutkivaa testausta tehdessä tulisi ottaa kantaa myös sovelluksen käytettävyyteen. Tutkivaa testausta suorittava henkilö voi kommentoida käytettävyyttä sessionsa jälkeen ja näin yrityksen tuotetta pystytään parantamaan osana testausta. Tätä ajatusta tukee tutkimustulos (Itkonen *et al*., 2007).

Järjestelmätestaustasolla tulee muistaa eri selaimilla testaus. Useiden työntekijöiden käyttäminen testaamisessa edistää myös sitä, että järjestelmää käytetään eri ympäristöissä. Näin testaukseen tulee luontevaa vaihtelua. Jokaisella testaajalla on lisäksi omat organisaationsa testiympäristössä, ja käytettävien tunnusten hallintaoikeudet vaihtelevat. Järjestelmätestaus toteuttaa samalla yhteensopivuustestausta, kun testausta tehdään eri selaimilla ja ympäristöillä.

Kuten testausstrategiaa suunniteltaessa todettiin, kannattavin lähestymistapa olemassa oleviin sovelluksiin on regressiotestauksen automatisointi järjestelmätestaustasolla. Regressiotestauksen toteuttamista on käsitelty alakohdassa 4.4.5 Testiautomaatio.

4.3.5 Hyväksymistestaus

Hyväksymistestaus jaetaan ensimmäisen versiojulkaisun testausta varten kolmeen eri osaan. 1) Testiautomaatiolla toteutetut hyväksymistestit. Käsin täydennettynä niiden testitapausten osalta, joita ei voida tai ole ehditty automatisoida. 2) Ainakin yksi sessio tutkivaa testausta testaajan suorittamana, mielellään myös jonkun muun. 3) Tietoturvaskannaus.

Selkeitä läpäisyehtoja hyväksymistestaukselle ei osattu asettaa testausstrategian luontivaiheessa. Tämä johtuu siitä, että kellään yrityksessä ei ole hyvää arviota paljonko testausta on yleensä tehty versiojulkaisuvaiheessa ja onko se ollut riittävää. Ehtojen luomiseen päätettiin palata ensimmäisen versiojulkaisun jälkeen, kun pystytään paremmin arvioimaan hyväksymistestausta. Läpäisyehtojen luontiin pitää osallistua ainakin teknologiapäällikön ja testaajan sekä mielellään myös muun tiimin, jotta kaikilla on tiedossa miten hyväksymistestausvaihe menee ja miten se vaikuttaa aikatauluihin. Myöhemmin hyväksymistestaukseen pyritään myös lisäämään ainakin suorituskykytestaus testiautomaatiolla toteutettuna.

Hyväksymistestaus suoritetaan Gpdemo-ympäristöön siirretyllä versiolla. Näin pystytään varmistamaan, että Beta-ympäristön vaikutukset järjestelmään pystytään sulkemaan pois ja järjestelmää saadaan testattua useammassa eri ympäristössä.

Hyväksymistestausvaiheessa suoritetaan myös tietty testitapausjoukko eri selaimilla. Testitapausjoukko kattaa kaikki eri moduulityypit ja testitapauksia on yhteensä 24, joilla jokaisella on 1-7 testitapausvaihetta. Osa näistä testeistä automatisoidaan seuraavassa alakohdassa esitetyn mukaan. Automatisoimalla osa hyväksymistestauksesta siitä saadaan kattavampi ja nopeampi suorittaa.

Lopullisen päätöksen hyväksymistestauksen valmistumisesta tekee teknologiapäällikkö. Testaaja raportoi hänelle hyväksymistestauksen tulokset, antaa kuvauksen tutkivan testauksen tuloksista sekä antaa oman arvionsa järjestelmän julkaisuvalmiudesta.

4.3.6 Testiautomaatio

Testiautomaatiolla haluttiin korvata paljon aikaa vieneet manuaalisesti suoritetut testitapaukset. Manuaalisiaa testejä käytettiin ennen varmistamaan, etteivät järjestelmän perustoiminnallisuudet lakanneet toimimasta päivitysten yhteydessä. Kuten testausstrategiaa suunniteltaessa todettiin, kannattavin lähestymistapa olemassa oleviin sovelluksiin on regressiotestauksen automatisointi järjestelmätestaustasolla.

Automaatiotestaustyökalun valinta oli yksi suurimmista päätöksistä prosessin aikana. Työkalun tärkeimmäksi ominaisuudeksi määritettiin usealla selailemalla saman testitapauksen ajaminen nopeasti ja helposti. Lisäksi uusien testitapausten luomisen piti olla helppoa myös uusille työntekijöille. Edullinen hinta oli myös tärkeä tekijä työkalun valinnassa. Kipurajaksi asetettiin noin 1000 dollaria vuodessa, riippuen työkalun ominaisuuksista. Tämä on kohtalaisen vähän eikä mahdollista monimutkaisemman työkalun valintaa vaan aiheuttaa sen, että valinnassa joudutaan tekemään kompromisseja.

Yrityksessä oli jo hylätty aikaisempien kokemusten perusteella yksi käytetyimmistä open source testiautomaatiotyökaluista, Selenium. Se oli todettu sopimattomaksi yrityksen tarpeisiin ja rajattu pois. Seleniumin käyttöä ei lähdetty siis lähdetty tutkimaan tämän työn puitteissa. Seleniumia olisi ehkä pitänyt harkita uudelleen, koska voi olla, että sen sopimattomuus johtui epäonnistuneesta käyttöönotosta ja Seleniumin uudemmat versiot olisivat voineet korjata aiemmin ilmenneet ongelmat.

Yrityksen kehittäjillä ja testaajalla ei ollut aikaisempaa kokemusta muista testiautomaatiotyökaluista, joten valintaa alettiin tekemään etsimällä internetin kautta eri työkaluja ja kokeilemalla niitä. Monet kaupalliset työkalut karsiutuivat pois liian korkean hinnan takia.

Testiautomaatiotyökaluksi valittiin SahiPro, joka on Tyto Softwaren testiautomaatiotyökalu. SahiPro on skriptien pohjalta toimiva käyttöliittymätestaustyökalu. SahiProssa on record-and-playback-ominaisuus, jota kritisoidaan jonkin verran testausalalla. SahiPro kuitenkin itse argumentoi sen puolesta: sillä pystytään nopeasti luomaan pohjakoodia, jota pystytään jälkeenpäin skriptaamaan halutulla tavalla. Heidän tarkoituksena ei ole pelkästään tarjota record-and-playback-ominaisuutta, vaan se toimii nopeuttavana työkaluna testien tekoon. SahiPro oli nopea ottaa käyttöön ja testien luonti sujui nopeasti. Testaajan ollessa osa-aikatyöntekijä nähtiin tarpeelliseksi valita työkalu, jolla päästään nopeasti testaamaan ja joka vastaa riittävän hyvin yrityksen tarpeisiin. (SahiPro, 2015)

4.3.7 Tietoturvatestaus

Tietoturva on tärkeä osa web-sovelluksia, kuten työssä on jo tuotu esille. Kohdejärjestelmää on testattu Acunetix-tietoturvaskannerilla sovelluksen kehityksen alusta asti. Acune-

tix ajaa kohdejärjestelmään testejä, joissa skanneri esiintyy kuin hakkeri. Acunetix pystyy käsittelemään myös AJAXia, joka oli tärkeä asia tietoturvatestaustyökalua valittaessa tähän kohdejärjestelmään. (Acunetix, 2015)

Acunetix-skanneri ajaa tietoturvatestit kerran viikossa kohdejärjestelmään. Skannauksen tuloksista tulee sähköposti testaajalle ja kehittäjille. Testaaja käy läpi tulokset ja ilmoittaa kehittäjille, mikäli skannauksessa on löytynyt uusia huomioitavia tietoturvariskejä.

Skanneri ajaa testit sellaisessa ympäristössä, johon sovelluksen uusin versio pitää päivittää erikseen. Tämä nähdään suurimpana ongelmana tietoturvatestauksen lähtötilanteessa. Skannerin tulisi ajaa testit viimeisimpään versioon, jotta tietoturvaongelmiin pystytään puuttumaan heti ja uusien ongelmien syiden selvittely helpottuu. Jos skannattavaan versioon on tehty paljon muutoksia, voi olla haastavaa löytää tietoturvaongelman syy nopeasti, ja tämä hidastaa kehitystyötä.

4.3.8 Suorituskykytestaus

Lähtötilanteessa järjestelmään ei tehty suorituskykytestausta millään työkalulla. Suorituskykyä arvioitiin tutkivan testauksen yhteydessä ja järjestelmää käytettäessä. Suorituskykytestauksen lähtökohtana oli luoda keinot testata yksittäisen käyttäjän yksittäisen sivun latausnopeus sekä raporttien luomisnopeus. Suorituskykytestauksen tuomista testausprosessiin mukaan lykättiin, koska resurssien puutteessa sen toteuttamiseen ei vielä pystytty perehtymään. Suorituskykytestaukseen on tarkoitus tutustua tarkemmin ja ottaa käyttöön kun muu testausprosessi on saatu kuntoon.

4.3.9 Käytettävyystestaus

Käytettävyystestausta ei tuoda testausstrategian alussa mukaan prosessiin. Käytettävyystestauksen suunnitteluun pitäisi tehdä paljon taustatutkimusta ja miettiä mitkä ovat yrityksen tarpeet, kun järjestelmä on kuitenkin jo olemassa oleva. Käytettävyystutkimus voidaan jatkossa tuoda mukaan testausstrategiaan.

4.4 Yhteenveto testausstrategiasta

Edellisen kohdan alussa kuvassa 3 kuvattiin testausstrategian suunnitelma ensimmäiseen versiojulkaisuun. Taulukossa 1 on esitetty eri vaiheet ja osa-alueet, joita testausstrategiassa on. Lisäksi taulukosta käy ilmi kuka on vastuussa mistäkin vaiheesta ja millä menetelmillä testausta tehdään ja milloin testausta tehdään. Testausstrategiassa käytetään eri testaustasoja useammassa vaiheessa hyödyksi. Lisäksi samaa testiautomaatiota hyödynnetään sekä järjestelmätestausvaiheessa että hyväksymistestausvaiheessa. Kun automaatiotestien määrää kasvatetaan, on mahdollista, että osa testeistä ajetaan vain hyväksymistestausvaiheessa.

Taulukko 1. *Eri testausvaiheiden/-toimintojen näkyminen testausstrategiassa*

Toiminto/Vaihe	**Taso**	**Menetelmä**	**Toteuttaja**	**Suoritus-aika**
Toiminnallinen testaus				
Kehitystestaus	Kehittäjän valittavissa	Kehittäjän valittavissa	Kehittäjä	Päivittäinen
Tikettitestaus	Järjestelmätestaus	Sessiopohjainen, rajattu tutkiva testaus	Testaaja, mahd. kehittäjä ja projektiassistentti	Päivittäinen
Regressiotestaus käyttöliittymän kautta	Järjestelmätestaus	Testiautomaatio	Testaaja (testiautomaatio)	Viikoittainen
Regressiotestaus käyttöliittymän kautta	Hyväksymistestaus	Testiautomaatio	Testaaja (testiautomaatio)	Ennen versiojulkaisua
Versioehdokkaan arviointi	Hyväksymistestaus	Tutkiva testaus	Testaaja	Ennen versiojulkaisua
Ei-toiminnallinen testaus				
Tietoturvatestaus	Järjestelmätestaus	Tietoturvaskanneri	Testaaja (tietoturvaskanneri)	Viikoittainen JA ennen versiojulkaisua
Suorituskykytestaus	-	-	-	Jatkokehitys
Käytettävyystestaus	-	-	-	Jatkokehitys

5. TESTAUSSTRATEGIAN KÄYTTÖÖNOTTO JA KEHITTÄMINEN

Tässä luvussa käsitellään miten suunniteltua testausstrategiaa sovellettiin kohdejärjestelmään. Testausstrategiaa sovellettiin kahteen eri versiojulkaisuun, jotka järjestelmästä tehtiin diplomityön tekemisen aikana. Testausstrategiaa on käsitelty eri testausvaiheiden mukaan. Ensimmäisen versiojulkaisun jälkeen jokaisen alakohdan lopussa on esitetty selkeät toimenpiteet, millä tavalla testausta kehitetään seuraavaan versioon. Toisen versiojulkaisun jatkokehitysideat ovat luvussa 6 samalla, kun eri testivaiheiden onnistumista ja tarpeellisuutta on arvioitu.

5.1 Versio 4.10, marraskuun versiojulkaisu

Ensimmäinen versio, johon testausstrategiaa sovellettiin, oli marraskuun versiojulkaisu eli version 4.10. julkaisu. Versiojulkaisu sisälsi 110 tikettiä, joista 57 oli vikakorjauksia ja 53 uusia ominaisuuksia. Suurimmat yksittäisen parannusosa-alueet olivat raportoinnin parantaminen sekä järjestelmän optimointi. Vikatiketeistä yksi oli blocker-tason korjaus ja 14 tikettiä oli critical-tason korjauksia. Muut olivat näitä prioriteetteja alempitasoisia. Korjaukset blocker-tiketteihin julkaistaan yleensä välittömästi pikakorjauksina, joten niiden osuus versiojulkaisussa pienin.

Alakohdissa on käsitelty jokaisen testauksen osa-alueen toteutuminen marraskuun versiojulkaisun testausprosessin aikana. Alakohdissa on ensimmäisenä tuotu esille, mikä oli osa-alueen tavoite testausstrategiaa luotaessa. Sen jälkeen on käsitelty miten tavoite toteutui. Samassa kohdassa on käsitelty mitä ongelmia ilmeni. Viimeiseksi jokaisessa alikohdassa on esitetty kehitystoimenpiteet seuraavan versiojulkaisun testausprosessiin.

5.1.1 Tikettitestaus

Tavoite: Jokainen uusi ominaisuus ja vikakorjaus testataan käsin tutkivaa testausta hyödyntäen.

Toteutuminen: Järjestelmätason testausta julkaisuversioon päästiin tekemään noin kaksi viikkoa ennen julkaisupäivää. Yhteensä tikettitestausta tehtiin 55 tuntia. Tällä saatiin katettua lähes kaikki vika- ja kehitystiketit, joita järjestelmäversioon tuli. Tarkoitus oli saada kaikki vika- ja kehitystiketit käytyä läpi tällä tavalla tutkivalla testauksella, mutta tähän oli liian vähän resursseja käytettävissä. Osa tiketeistä jouduttiin toteamaan valmiiksi ilman, että niille tehtiin erillistä testausta ajanpuutteen takia.

Ongelmaksi tikettitestauksessa nousi dokumentaation puute. Kehitys- ja vikatiketeissä ominaisuuksia ei ollut kuvattu riittävän tarkasti, ja tikettien testaajat joutuivat kysymään kehittäjiltä, miten ominaisuuden tulisi toimia. Tällöin sekä testaajan että kehittäjän työ keskeytyi. Lisäksi testaajilla meni paljon aikaa löytää tarvittavat ominaisuudet järjestelmästä. Tämä johtui siitä, että kuvausta perusominaisuuksien toimintatavasta ei ollut. Esimerkiksi yhdeltä testaajalta meni yhtenä päivänä tehdystä työstä puolet siihen, että hän selvitti miten ominaisuudet toimivat. Tähän ongelmaan on esitelty ratkaisuehdotus jatkossa.

Lisäksi ongelmana oli, että varsinaisen testaajan sairasloman vuoksi tikettitestausta ei päästy suorittamaan kunnolla kun vähän ennen version valmistumista. Seuraavaan versioon tikettitestaus tulisi toteuttaa niin, että sitä tehdään jatkuvasti uusien tikettien valmistuttua.

Kehitystoimenpiteet: Tikettitestausta tehdään uuden version kehityksen alusta asti heti kun testattavia tikettejä on tarjolla.

5.1.2 Hyväksymistestaus

Tavoite: Valmiiseen julkaisuversioon ajetaan joukko automaatiotestejä ja tehdään vapaata tutkivaa testausta ennen kuin versio ilmoitetaan julkaisukelpoiseksi.

Toteutuminen: Tutkivalle testaukselle ei juurikaan jäänyt aikaa marraskuun versiojulkaisussa. Testausaika käytettiin järjestelmällisesti kehitys- ja vikatikettien testaamiseen.

Seuraavaan versioon hyväksymistestausvaiheen tutkivan testauksen osuutta tulee lisätä. Vaikka jokainen vika- ja kehitystiketti tulisi tutkia erikseen ja in testing-testaus on kattavaa, tulisi hyväksymistestausvaiheessa järjestelmää tarkastella kokonaisuutena ja miettiä erilaisia ominaisuuksia, joihin muutokset ovat saattaneet vaikuttaa, ja testata näitä. Lisäksi tulee varmistaa, että tärkeimmät ominaisuudet varmasti toimivat.

Hyväksymistestauksen lopulliset manuaalitestit suoritettiin Gpdemo-ympäristössä suunnitelman mukaisesti. Aiemmin työssä tuotiin esille tarve hyväksymistestauksen automatisoinnista. Tähän versioon järjestelmän hyväksymistestaus jouduttiin kuitenkin suorittamaan käsin. Tämä vei paljon aikaa, ja hyväksymistestauksen automatisointia tulisi harkita. Tähän vaikuttaa se, että järjestelmälle seuraavaan versioon ajetaan järjestelmätestaustasolla automaatiotestejä viikoittain, jolloin hyväksymistestausta on perusteltua suorittaa käsin.

Hyväksymistestauksen lisäksi järjestelmälle suoritettiin yksi parin tunnin sessio vapaata tutkivaa testausta julkaisuversiossa. Tässä sessiossa ei järjestelmästä löytynyt vikoja, vaan ainoastaan pieniä raportointiin liittyviä käytettävyysongelmia.

Kehitystoimenpiteet: Varataan hyväksymistestaukselle enemmän aikaa ennen seuraavaa versiojulkaisua, jotta alkuperäistä suunnitelmaa saadaan käytettyä testausstrategian mukaisesti.

5.1.3 Testiautomaatio

Tavoite: Korvataan entiset manuaaliset regressiotestit testiautomaatiolla.

Toteutuminen: Testiautomaatiolla oli tarkoitus toteuttaa kohdan 4.3.5 mukaisesti jatkuvaa järjestelmän testausta eli regressiotestausta, ennen kokonaisvaltaisempaa järjestelmätestausta. Valittu testiautomaatiotyökalu ei kuitenkaan toiminut odotetusti ja testejä ei päästy automatisoimaan, vaan ne piti ajaa käsin järjestelmään eikä tätä kehitysvaiheessa tehty. Testaajan työpanos ei ollut käytettävissä marraskuun aikana eikä kehittäjillä ollut aikaa perehtyä automaatiotestien suorittamiseen.

Automaatiotestejä tehtiin yhteensä 15 kappaletta, jotka kattoivat noin 50% entisistä manuaalitesteistä, jotka automaatiotestien oli tarkoitus korvata.

Testiautomaatiotyökalu SahiPron kanssa tuli ongelmia, kun työkalun versiopäivityksen yhteydessä osa automaatiotesteistä oli käyttökelvottomia. Sen vuoksi erästä olennaista lomakekenttää ei voinut automaatiotesteillä enää täyttää Internet Explorer ja Firefox–selaimilla. Tätä ongelmaa ei pystytty ratkaisemaan ennen marraskuun versiojulkaisua. Tällä tavalla suurin osa testeistä oli suorituskelvottomia mainituilla selaimilla.

Automaatiotestit saatiin kuitenkin suoritettua Chrome-selaimella. Tämä jo selvästi vähensi käsin tehtävää työtä. Automaatiotesteillä ei havaittu yhtään vikaa järjestelmästä. Varmuuden vuoksi automaatiotestit suoritettiin myös käsin eikä tälläkään tavalla löydetty vikoja testien kattamista ominaisuuksista.

Seuraavaan versioon automaatiotestien määrää tulee nostaa. Uusia automaatiotestejä tulisi luoda viikoittain, jotta niillä saataisiin parempi kattavuus järjestelmään. Lisäksi automaatiotestit tulisi saada viikoittaiseen ajoon, jolloin järjestelmään tulleet viat havaittaisiin varhaisemmassa vaiheessa. Muut työntekijät tulee perehdyttää automaatiotestityökalun käyttöön ja tulosten tulkitsemiseen.

Kehitystoimenpiteet: Ajastetaan automaatiotestit suoritettavaksi viikoittain. Luodaan lisää testitapauksia, jotka toteutetaan testiautomaatiolla.

5.1.4 Tietoturvatestaus

Tavoite: Tietoturvaskannausta jatketaan niin kuin ennenkin: kerran viikossa suoritetaan tietoturvaskannaus.

Toteutuminen: Acunetixiin ei tehty muutoksia. Kerran viikossa Acunetix ajaa tietoturvatestit järjestelmälle kolmella eri profiililla. Tämä on koettu riittävän kattavaksi ja järjestelmästä ei ole löytynyt sellaisia ongelmia, joita Acunetix olisi voinut havaita. Acunetix lähettää testit ajettuaan tulokset kehittäjien ja testaajan sähköpostiin. Testaaja käy läpi tietoturvaskannauksen tulokset ja ilmoittaa mahdollisista tuloksissa ilmenneistä muutoksista kehittäjille.

Uuden version viimeisimmässä Acunetix-ajossa ilmeni 10 eri ongelmaa, joista yksi oli high-level tason uhka ja seitsemän Informational-level tason uhkaa. Vakavin ongelmista oli BREACH-hyökkäykselle alistuminen. Tämä ongelma on ollut tiedossa pitkään, mutta se ei muutoksista huolimatta ole poistunut. Ongelma on listattu seuraavaan versiojulkaisuun korjattavaksi.

Kehitystoimenpiteet: Päätetään toimintatavoista, jolla määritetään kuka ja milloin päivittää ympäristön, johon tietoturvaskanneri suorittaa skannauksen.

5.1.5 Suorituskykytestaus

Tavoite: Suorituskykytestausta tutkitaan enemmän, jos aikaa on.

Toteutuminen: Suorituskykytestausta ei erikseen suoritettu. Suorituskykyä arvioitiin muutaman kehitystiketin kautta, joissa oli nopeutettu sivulatauksia. Nämä arvioitiin silmämääräisesti, eikä sivulatauksista mitattua dataa tarkkailtu.

Eräs asiakkaalta tullut kehitystiketti liittyi järjestelmän suorituskykyyn. Organisaatiossa oli paljon toteutuksia ja tarkastelukohteita, ja siellä oli huomattu, ettei AJAXilla toteutettu ”käyttäjän valinta” -kenttä toiminut. Tämän ongelman ilmennettyä kiinnitettiin huomiota suorituskykyyn, ja järjestelmään tehtiin sitä tehostavia muutoksia.

Suorituskykytestauksessa voitaisiin hyödyntää käytettävissä olevaa tapahtumalokia. Tällä tapahtumalokilla voitaisiin luoda eri käyttötapauksia ja tarkastella, kuinka paljon mihinkin askeleeseen meni aikaa. Sitten tätä tulosta voitaisiin verrata uusien ominaisuuksien ja vikakorjausten jälkeen tehtyihin sivulatauksiin, ja pysyttäisiin tarkastelmaan onko sivulatauksissa tapahtunut muutosta.

Kehitystoimenpiteet: Mahdollisuuksien mukaan testaaja käyttää työaikaansa tutustumalla siihen, miten testiautomaatiotyökalua voitaisiin käyttää suorituskykytestaukseen tässä web-sovelluksessa.

5.1.6 Käytettävyystestaus

Tavoite: Resurssien puitteissa tutkitaan tarkemmin, jos mahdollista.

Toteutuminen: Resurssien puutteen vuoksi testaajaa ei ollut mahdollista kouluttaa käytettävyystestaamiseen. Käytettävyys otettiin huomioon jo suunnitteluvaiheessa, ja kaikki järjestelmän kehittäjät kiinnittävät huomiota järjestelmän käytettävyyteen. Tällä tavalla ei yksittäisiä erikoistapauksia saada arvioitua ja parannettua, mutta käytettävyyttä pystytään arvioimaan melko hyvin. Koska järjestelmällä ei ole muita käyttäjiä kuin ennalta tiedetyistä maista, voidaan uuden asiakkuuden myötä arvioida tuleeko asiakkaan kansalaisuuden perusteella tehdä erityistä konfiguraatiota asiakasta varten. Näihin erityistapauksiin voidaan kiinnittää huomiota tarvittaessa, koska asiakkaiden käyttömaat ovat tiedossa.

Kehitystoimenpiteet: Ei toimenpiteitä.

5.1.7 Dokumentaatio

Tavoite: Ei asetettu alun perin. Ongelmat huomattiin testausprosessin yhteydessä.

Jo testausstrategiaa luotaessa todettiin, että dokumentaation puute on iso ongelma testauksen toteuttamiselle. Uutta testausstrategiaa ensimmäistä kertaa käytettäessä huomattiin, että tutkivan testauksen hyödyntämisestä huolimatta dokumentaation puute aiheutti isoja ongelmia.

Kuten alakohdassa 5.1.2 kävi ilmi, testaajien ja kehittäjien työtä hidasti puuttuva dokumentaatio ja kehitys- ja vikatikettien huono laatu. Tikettien laatuun tulisi panostaa, kun testausta halutaan kehittää seuraavaan versioon. Tällöin testaajat voivat keskittyä testaamiseen ja kehittäjien työ ei keskeydy koko ajan.

Erityisen tärkeän dokumentaation parantamisesta tekee se, että yrityksen teknologiapäällikkö muuttaa ulkomaille pariksi kuukaudeksi. Hänen ollessaan toimistolla kehittäjät ja testaaja kysyvät häneltä paljon tarkennuksia eri asioihin. Teknologiapäällikön kanssa kommunikointi tapahtuu jatkossa yrityksen keskusteluohjelman, Skypen tai sähköpostin kautta.

Yrityksessä ei ole dokumentaatiota, koska sen ylläpitäminen on koettu raskaaksi ja kehitystiimi on ollut pieni. Pieneen ja kohtalaisen pysyvään kehitystiimiin on kertynyt paljon hiljaista tietoa järjestelmästä. Yrityksen tapa dokumentoida asiat tiketteihin on ongelmallinen testauksen näkökulmasta. Samasta ominaisuudesta voi olla monta eri tikettiä, ja ne sisältävät kukin eri tietoja toiminallisuudesta. Tikettejä ei välttämättä ole liitetty toisiinsa, joten testaajan käytettävissä ei ole kattavaa dokumentaatiota toiminnallisuuden ominaisuuksista ja toimintatavasta.

Vika- ja kehitystiketeissä käytetään sekaisin englanninkielisiä ja suomenkielisiä termejä, eikä termien selityksiä löydy mistään. Kehittäjät eivät kuvaa tiketeissä toiminnallisuutta

yleensä kovinkaan tarkasti. Tästä aiheutuu se, että testaajan pitää itse selvittää, miten perusominaisuuden tulisi toimia. Ongelmana on, että testaajat joutuvat jatkuvasti tarkastamaan asioita kehittäjiltä ja keskeyttämään sekä oman että kehittäjien työn.

Yhtenä ratkaisuna dokumentaatioon voisi kokeilla JIRAan liitettävää lisäosaa, joka mahdollista dokumentaation tekemisen. Lisäosan avulla järjestelmään voidaan luoda sanastoa, jonka avulla eri tiketeissä voidaan tarkastella sanan merkitystä viemällä hiiri sanan päälle. Tällä tavalla pystyttäisiin luomaan dokumentaatio ja ylläpitämään sitä siinä järjestelmässä, jota kehittäjät, testaajat ja yrityksen johto käyttävät päivittäin. Eri ominaisuuksien toiminta voitaisiin tällöin kirjoittaa yhden kerran, ja sen saisi tarkastettua eri tikettien yhteydessä. Sanastoa voisi myös luoda tarpeen tullessa helposti lisää päivittäisessä käytössä. Tällaista lisäosaa ei välttämättä vielä ole JIRAan, eikä sen toteutuskuluista ole arviota.

Toinen ratkaisu dokumentaatioon olisi wikin luominen. Wikissä asioiden linkittäminen toisiinsa olisi helppoa hyperlinkkien avulla. Eräs tällainen wikin luomisen mahdollisuus olisi luoda yrityksellä jo käytössä olevaan Confluenceen parempi dokumentaatio. Confluence on Atlassian-ohjelmistoyrityksen organisaatiowikiohjelmisto. Confluence ja JIRA tarjoavat jo nyt mahdollisuuden linkittää Confluence sivuja JIRA:n tiketteihin ja toisin päin. Sivujen linkittämistä ei ole toteutettu hyvin ja se on hieman kömpelöä. Linkittämisessä olisi se etu, että Confluence-sivulle, joka kuvaa tiettyä ominaisuutta, voitaisiin lisätä jokainen tiketti, joka liittyy ominaisuuteen. Tällä tavalla saataisiin kattava dokumentaatio tehtyä yhdistäen jo käytössä olevat järjestelmät. Wikin luomiseen pitäisi resursoida työtunteja, mutta uusia hankintoja ei tarvitsisi tehdä. Jos dokumentaatiossa päädyttäisiin wikin tekemiseen, tärkeää olisi wikin päivittäminen ja aktiivinen käyttö kaikkien osalta.

Kehitystoimenpiteet: Päätetään yhdessä teknologiapäällikön kanssa aletaanko järjestelmän dokumentaatiota kehittämään. Tutustutaan tarkemmin eri vaihtoehtoihin ja pohditaan tiimin kanssa, mikä olisi järkevä ratkaisu parantaa dokumentaatiota.

5.1.8 Muu testausprosessi

Tavoite: Ei asetettu alun perin. Ongelmat huomattiin testausprosessin yhteydessä.

Testausta suoritettaessa huomattiin, että koko testausprosessista muodostui ongelmallinen. Testausstrategiaa ei ollut mahdollista soveltaa suunnitellusti, eikä siitä tullut tavoitellun tehokas. Tähän oli kaksi pääsyytä: dokumentaation puut sekä testaajien ja kehittäjien työn jatkuva keskeytyminen. Nämä syyt liittyvät toisiinsa oleellisesti.

Sekä yksi testaajien että kehittäjien työtä keskeyttävä tekijä oli testausympäristön puutteellisuus. Testaamisen yhteydessä nimittäin huomattiin, että testaajat eivät pysty teke-

mään tietokantaan muutoksia, joita joidenkin testien suorittaminen vaatisi. Tällöin testaaja joutui keskeyttämään testauksena ja pyytämään kehittäjältä muutosta tietokantaan. Tämä keskeytti samalla kehittäjän työn. Kehityksenä seuraavaan versioon testaajille pitäisi antaa oikeudet tehdä muutoksia beta-ympäristön tietokantaan. Tällöin kumpikaan, testaaja tai kehittäjä, ei joudu keskeyttämään työtään.

Mikäli testaajille ei voida antaa oikeuksia tietokannan muokkaamiseen, tulisi muutoksia varten luoda toimintakäytäntö. Käytäntö voisi esimerkiksi olla, että testaajan pyytäessä muutosta kehittäjien chat-huoneessa ensimmäinen kehittäjä, jolla olisi sopiva tilanne, hoitaisi muutoksen. Tämä aiheuttaa sen, että testaajan pitää keskeyttää testauksensa siksi aikaa ja siirtyä eri tehtävään, josta aiheutuu taas jo mainittua työn keskeytymistä, mikä ei ole järkevää muutenkin pienien testausresurssien kannalta.

Kehitystoimenpiteet: Keskustellaan kehitystiimin kanssa yrityksen prosesseista.

5.2 Versio 5.1, maaliskuun versiojulkaisu

Järjestelmän versio 5.1 eli maaliskuussa julkaistu versio toi järjestelmään ison käyttöliittymäuudistuksen. Tämä uudistus vaikutti todella merkittävästi järjestelmän testaamiseen ja suunniteltua testausstrategiaa ei tässäkään versiossa päästy kunnolla soveltamaan. Tämän kohdan aliluvuissa on käyty edellisen kohdan mukaisesti läpi eri testaustavat, joita järjestelmälle tehtiin.

Käyttöliittymäuudistus valmistui kunnolla testattavaksi vasta huhtikuun toisella viikolla, jolloin käyttöliittymään tehtiin viimeiset muutokset. Käyttöliittymäuudistus myös aiheutti sen, että monia eri ominaisuuksia ei ollut järkevää testata tai testaaminen oli mahdotonta aikaisemmassa vaiheessa.

Uudistuksen tarkoitus oli tehdä järjestelmästä responsiivinen ja mahdollistaa järjestelmän mobiilikäyttö. Tämä toikin testausstrategiaan paljon uutta. Mobiililaitteilla piti tehdä paljon testausta ja mobiilialustoilla ilmeni paljon ongelmia.

Aliluvut on jaoteltu samalla tavalla kuin viime kohdassa. Jokainen alikohta sisältää viime versiojulkaisussa esitetyt jatkokehitystoimenpiteet, niiden toteutumisen ja lyhyen tähtäimen jatkotoimenpiteet seuraavaan versiojulkaisuun. Pitemmän ajan jatkokehitysideat yritykselle on esitelty kohdassa 6.3.

5.2.1 Tikettitestaus

Tavoite/Asetetut kehitystoimenpiteet: Tikettitestausta tehdään alusta asti heti, kun testattavia tikettejä on tarjolla.

Toteutuminen: Tikettitestausta alettiin tekemään heti, kun testattavia tikettejä oli tarjolla, ja niitä pystyi testaamaan, mutta prosessissa ilmeni useampia ongelmia. Kaikki tiketit saatiin testattua, joskin osa testauksesta jouduttiin tekemään kiireellä.

Kehittäjät työstivät tammikuussa suuritöisempiä tikettejä, ja helmikuun lopulla tehtiin kerralla suuri määrä vähätöisempiä tikettejä testaukseen. Työmäärä jakautui siis testaajan näkökulmasta epätasaisesti. Dokumentaation parantamiseksi ei tehty toimenpiteitä, joten testaajien vaikeudet testata tikettejä jatkuivat. Järjestelmään tehtiin käyttöliittymämuutosta, ja jotkut perustoiminnallisuudet käyttöliittymästä olivat pitkään rikki, eikä testausta saanut tehtyä.

Maaliskuun versiojulkaisuun saatiin tikettitestaukseen käytettyä hieman enemmän aikaa kuin marraskuun versiojulkaisuun. Kaikki vika- ja kehitystiketit saatiin testattua, mutta muutaman isomman ominaisuuden testaamisen koettiin jääneen liian pintapuoliseksi ajanpuutteen takia. Viime versiossa useampi henkilö osallistui tikettitestaukseen, mutta tässä versiossa testaaja suoritti kaiken tikettitestauksen. Tämä rajoitti hieman eri ympäristöillä testaamista. Kuitenkin testaajalla oli käytössä mobiililaitteiden myötä useampi ympäristö, joka tasoitti testausympäristöjen yksipuolisuutta.

Kehitystoimenpiteet: 1) Pyritään ottamaan tikettien toteutusjärjestyksessä huomioon myös testaaja ja aikatauluttamaan tehtävät työt paremmin. 2) Järjestetään tikettien kirjoittamisesta koulutus, jotta testaajat eivät joudu jatkuvasti keskeyttämään kehittäjien työtä.

5.2.2 Hyväksymistestaus

Tavoite/Asetetut kehitystoimenpiteet: Varataan hyväksymistestaukselle enemmän aikaa ennen seuraavaa versiojulkaisua, jotta alkuperäistä suunnitelmaa saadaan kokeiltua kunnolla.

Toteutuminen: Hyväksymistestaus jäi kiireen takia hyvin pintapuoliseksi. Järjestelmän julkaisu myöhästyi yli kuukaudella, ja kunnon hyväksymistestausvaihe ohitettiin lähes kokonaan.

Ongelmat testiautomaatiotyökalun käytössä on kuvattu seuraavassa alakohdassa. Koska testiautomaatiotyökalu ei ollut käytettävissä, niin hyväksymistestausta jouduttiin tekemään käsin. Manuaaliset testit suoritettiin vain yhdellä selaimella järjestelmään, ja tämäkin tapahtui sen jälkeen, kun versio oli jo asiakkaiden saatavilla joissain ympäristöissä. Manuaalisissa testeissä löydettiin järjestelmästä muutama ongelma, joista yksi piti korjata pikakorjauksena.

Kehitystoimenpiteet: Yritys sitoutuu hyväksymistestausvaiheeseen ja varaa sille tarvittavan ajan, jotta hyväksymistestaus voidaan tehdä strategiassa määritetyllä tavalla, ja arvioimaan miten sitä pitäisi kehittää.

5.2.3 Testiautomaatio

Tavoite/Asetetut kehitystoimenpiteet: Ajastetaan automaatiotestit suoritettavaksi viikoittain. Luodaan lisää testitapauksia, jotka toteutetaan testiautomaatiolla.

Toteutuminen: Automaatiotestejä ei päästy järjestelmään ajamaan ollenkaan. Tämä oli suuri heikkous testauksessa ja turhaa käsityötä syntyi paljon testauksen aikana. Automaatiotestejä ei ollut järkevä päivittää keskeneräiseen käyttöliittymään, koska muutoksia tehtiin paljon nopealla tahdilla, ja automaatiotestien jatkuva päivittäminen olisi vienyt paljon resursseja, mutta tuonut vain vähän hyötyä. Automaatiotesteillä ei myöskään olisi pystytty ottamaan kantaa käyttöliittymän toimivuuteen.

Kehitystoimenpiteet: Päivitetään testiautomaatiotestit toimimaan uuden käyttöliittymän kanssa.

5.2.4 Tietoturvatestaus

Tavoite/Asetetut kehitystoimenpiteet: Päätetään toimintatavoista, joilla määritetään kuka ja milloin päivittää tietoturvaskannerin ympäristön version.

Toteutuminen: Erillistä päätöstä ei tehty, mutta testaaja piti huolen järjestelmän päivittämisestä tasaisin väliajoin. Tietoturvaskannerilla jatkettiin viikoittaista skannausta, eikä järjestelmästä löytynyt uusia ongelmia.

Järjestelmälle tehtiin version kehityksen aikana ulkopuolisen tahon suorittama ”tietoturvatutkinta”, joissa havaittiin joitakin ongelmia järjestelmässä. Raportin mukana saatiin selkeät ohjeet, miten tietoturvaongelmat oli huomattu, ja näitä päästiin testaamaan ohjeiden mukaisesti. Järjestelmä sai huhtikuussa tietoturvasertifikaatin.

Kehitystoimenpiteet: Luodaan toimintaohjeet skannausympäristön päivittämiseen.

5.2.5 Suorituskykytestaus

Tavoite/Asetetut kehitystoimenpiteet: Jos aikaa on, niin tutustutaan siihen, miten testiautomaatiotyökalua voidaan käyttää suorituskykytestauksessa tähän sovellukseen.

Toteutuminen: Järjestelmään tehtiin automaatiotestien avulla suorituskykytestausta. Automaatiotesteillä luotiin testitapaus, jossa tehtiin useampi sivunlataus. Tämä testitapaus ajettiin 20 kertaa testaajan työpäivinä järjestelmään. Alkuun testitapausta ajettiin monta kertaa päivässä, jotta saatiin vertailudataa erilaisilla kuormituksilla, ja nähtiin niiden vaikutus ajoaikaan. Tämän testitapauksen ajoaika kerättiin Excel-tiedostoon, jossa tarkkailtiin, paljonko testin ajoajan keskiarvo on per ryhmäajo ja tarkasteltiin kasvaako ajoaika merkittävästi jonkin päivityksen jälkeen. Testiautomaatiotyökalun tarjoamat ominaisuudet varmasti tarjoavat paremman tavan tehdä suorituskykytestejä, ja niihin tulisi perehtyä.

Tätä menetelmää ei voitu järkevästi soveltaa tässä versiojulkaisussa, koska automaatiotestejä ei päästy päivittämään käyttöliittymäuudistuksen myötä.

Kehitystoimenpiteet: Perehdytään paremmin testiautomaatiotyökalun tarjoamiin suorituskykytestauksen mahdollisuuksiin.

5.2.6 Käytettävyystestaus

Tavoite/Asetetut kehitystoimenpiteet: Ei toimenpiteitä.

Toteutuminen: Jokainen kehittäjä ja testaaja kiinnittivät erityishuomioita uuden käyttöliittymän käytettävyyteen version kehitysvaiheessa. Järjestelmää päivitettiin tiheään tahtiin niin, että uusin versio oli koko kehitystiimin saatavilla. Näin eri käyttöliittymäongelmat eri selaimilla löydettiin nopeasti. Useat työntekijät pystyivät arvioimaan käyttöliittymän toimimista jo kehitysvaiheessa.

Käyttöliittymästä kerättiin myös palautetta muutamalta asiakkaalta, joille näytettiin kehitysversiota uudesta käyttöliittymästä ja heiltä pyydettiin kommentteja. Palaute oli positiivista, ja eräs asiakas kysyi, oliko yrityksen palkattu käytettävyysammattilainen.

Kehitystoimenpiteet: Ei toimenpiteitä.

5.2.7 Dokumentaatio

Tavoite/Asetetut kehitystoimenpiteet: Päätetään yhdessä teknologiapäällikön kanssa, aletaanko järjestelmän dokumentaatiota kehittämään. Tutustutaan tarkemmin eri vaihtoehtoihin ja pohditaan tiimin kanssa, mikä olisi järkevä ratkaisu parantaa dokumentaatiota.

Toteutuminen: Dokumentaatiota ei parannettu edellisen version jäljiltä. Dokumentaation parantamiseen ei ollut käytössä resursseja kahdesta syystä. Maaliskuun versio sisälsi paljon isoja muutoksia järjestelmään. Ennen isojen muutosten toteuttamista dokumentaation kehittämisen aloittaminen ei ollut perusteltua. Dokumentaation puute ei merkinnyt tämän version testauksessa niin paljon kuin edellisen version, koska iso osa testauksesta oli käyttöliittymäuudistukseen keskittyvää eikä toiminnallisuuksien muuttumisen testausta. Kuten aiemmissa kohdissa on korostettu, käyttöliittymästä kokeiltiin eri versioita ja sitä hiottiin mahdollisimman pitkään, eikä siitä olisi ollut järkevää työstää dokumentaatiota sen ollessa vielä kesken.

Kehitystoimenpiteet: Jatkokehitys sama kuin asetetut kehitystoimenpiteet, koska muutosta ei tapahtunut.

5.2.8 Muu testausprosessi

Tavoite/Asetetut kehitystoimenpiteet: Keskustellaan kehitystiimin kanssa yrityksen prosesseista.

Toteutuminen: Aikatauluongelmien takia kehityskeskustelua ei pidetty.

Versio sisälsi isoja muutoksia ja testauksen kannalta tammi- ja helmikuu olivat hiljaisia, vaikka käytössä oli tikettien ”in testing”-tilaan siirtäminen niiden valmistuttua. Automaatiotestien lisäämisestä ei olisi ollut hyötyä, koska ne olisi kuitenkin pitänyt päivittää version valmistuttua.

Uutena testaamiseen tuli eri laitteilla testaaminen. Ennen järjestelmää testattiin edes jollain tasolla viidellä eri selaimella, joista yhdestä selaimesta oli käytössä 3 eri versiota. Testialustoiksi tuotiin mobiilikäyttöä tukevan käyttöliitymäuudistuksen vuoksi 3 eri tablettitietokonetta sekä yrityksen työntekijöiden kännykät. Tämä hidasti testaamista verrattuna entiseen, kun sama ominaisuus piti testata useammalla eri laitteella ja selaimella. Kehitystyötä ja testaamista vaikeutti se, kuinka eri tavalla järjestelmä toimi eri selaimilla ja alustoilla. Bugien todentaminen ja virheiden löytäminen oli haastavampaa. Esimerkiksi jonkin selaimen ongelman korjaaminen saattoi aiheuttaa ongelmia toisessa selaimessa.

Testausprosessi vaati paljon vuorovaikutusta kehittäjien ja testaajien välillä, koska muutoksia oli tehtävä nopealla aikataululla. Tämä saattoi parantaa muutoksien laatua, kun erilaisista ratkaisuista keskusteltiin kehitysvaiheessa ja ongelmakohtia pohdittiin useammasta näkökulmasta.

Kehitystoimenpiteet: Keskustellaan yrityksen prosesseista kehitystiimin kanssa.

6. ARVIOINTI JA JATKOKEHITYS

Tässä luvussa arvioidaan testausstrategiaa ja käsitellään testausstrategian kehittämistä jatkossa. Luvussa ehdotetaan, mitä konkreettisia toimenpiteitä yrityksen tulisi tehdä, jotta kehitys- ja testausprosessia saataisiin tehostettua.

6.1 Testausstrategian arviointi

Kun testausstrategiaa luotiin ja otettiin käyttöön, se vaikutti mukautuvan hyvin yrityksen prosesseihin. Se vastasi yrityksen tarpeisiin keveytensä ja helpon käyttöönoton takia. Se ei lisännyt kehittäjien työtaakkaa, joka oli tärkeä kriteeri yritykselle. Suunnitelman avulla saatiin tehtyä muutoksia, jotka selvästi paransivat testausta.

Testausstrategiaa ei kuitenkaan saatu vietyä suunnitelman mukaisesti läpi. Testaukselle jäi molemmissa versiojulkaisuissa liian vähän aikaa. Toiminnallisuuksien toteuttaminen venyi molemmissa versiojulkaisuissa suunnittelusta julkaisupäivästä seuraavan kalenterikuukauden puolelle, ja yrityksellä oli painetta saada versio julkaistua nopeasti. Tämä aiheutti sen, ettei tutkivalle testaukselle jäänyt juuri lainkaan aikaa. Tutkivaa testausta suoritettiin tikettitestauksessa, mutta valmiin järjestelmän laajempi tutkiva testaus varmistaisi sen, että tehdyt muutokset eivät ole vaikuttaneet arvaamattomiin osiin järjestelmässä. Järjestelmän asetusten, moduuleiden ja ominaisuuksien määrä on niin suuri, että riittävän kattava testaaminen vaatii aikaa.

Järjestelmään on pystyttävä luottamaan, koska järjestelmän monimutkaisuuden takia kaikkia mahdollisia tilanteita ei pystytä testaamaan ja jossakin vaiheessa versio on julkaistava. Yritys pystyy tekemään nopeasti asiakkaille pikakorjausversioita versiojulkaisun jälkeen, mutta yrityksen maine kärsii tästä.

Testausstrategia oli hyvin yksilöity yrityksen silloiseen testausprosessiin ja annettuihin testausresursseihin. Resurssien lisäksi toinen testausstrategiaa paljon määrittänyt seikka oli, että järjestelmä oli valmis kun testausstrategiaa ruvettiin luomaan siihen. Valmiiseen järjestelmään yksikkötestien ja sitä kautta integraatiotestien tuominen voi olla hyvin työlästä ja kallista.

Prosessin aikana opittiin paljon ohjelmistoalan ja testauksen perusongelmista. Automaatiotestien ajan tasalla pitämiseen ja työkalujen päivityksiin täytyy varata resursseja, mikäli niiden hyödyllisyys halutaan taata. Lisäksi prosessissa huomattiin, että varsinkin pienissä yrityksissä yhden henkilön merkitys projektille on suuri: testaajan joutuessa sairaslomalle ennen marraskuun versiojulkaisua testauksen suunnitelman mukainen läpivienti epäonnistui.

Testausstrategia kuitenkin vei yrityksen testausprosessia eteenpäin ja paransi järjestelmän laatua. Testiautomaation epäonnistunut käyttöönotto oli isoin ongelma testausstrategian käyttöönotossa. Tavoitteena oli, että automaatiotestejä olisi päästy hyödyntämään ensimmäisessä versiossa ja niitä olisi saatu kehitettyä seuraavaan versioon.

Web-sovellusten testaamiseen ja testitapausten luomiseen löytyy paljon erilaisia tekniikoita. Lisäksi web-sovellusten testaamiseen on paljon erilaisia testikehyksiä. Näitä ei tämän työn puitteissa päästy käsittelemään ollenkaan, koska testausstrategiassa keskityttiin luomaan yksinkertainen ja helposti käyttöönotettava strategia. Kun riittävät testauskäytännöt on saatu yrityksessä vakiinnutettua, yrityksen testaaja voi tutustua erilaisiin testaustekniikoihin ja –menetelmiin.

Vaikka testausstrategian käyttöönotossa oli ongelmia, suurin hyöty testausstrategian luomisesta oli siinä, että yrityksen sekä testauprosessin että muiden prosessien ongelmia pystyttiin tunnistamaan paremmin ja näihin ongelmiin pystytään jatkossa etsimään ratkaisuita.

6.2 Testausstrategian käyttöönoton arviointi

Testausstrategian tuominen osaksi yrityksen kehitysprosessia on iso muutos, johon yrityksen tulisi sitoutua, jos testausstrategian käyttöönotossa halutaan onnistua. Testausstrategian kehittämiseen tulisi useamman henkilön panostaa ja se pitäisi nähdä osana koko kehitysprosessia eikä nähdä vain irrallisena osana muusta kehitystyöstä.

Tätä testausstrategiaa luotaessa testaaja, teknologiapäällikkö ja kehittäjät eivät ymmärtäneet, miten paljon resursseja testausstrategian käyttöönotto ja testauksen kehittäminen olisi vaatinut jokaiselta kehitystiimin jäseneltä. Muutokseen ja aikatauluihin sitoutuminen olisi antanut aikaa kehittää testausstrategiaa, nyt versiojulkaisuiden venyessä testausstrategiaa ei päästy kunnolla toteuttamaan eikä varsinkaan kehittämään.

Yksi tämän diplomityön tärkeimpiä löydöksiä oli se, että vaikka testausstrategia suunniteltaisiin onnistuneesti ja yrityksen resursseihin sopivaksi, sen käyttöönotossa ja kehittämisessä voidaan epäonnistua, elleivät kaikkien sidosryhmien odotukset strategian suhteen ole yhtenäiset. Testausstrategiaa luotaessa ja käyttöön otettaessa tulee kiinnittää huomiota siihen, että odotukset resurssien, aikataulun ja teknologioiden suhteen ovat eri osapuolten välillä yhtenäiset. Testaaja ei pysty toteuttamaan strategiaa, mikäli projektin vastuuhenkilö ei varaa riittävästi resursseja ja aikaa testaukseen. Vastaavasti projektin vastuuhenkilö ei pysty aikatauluttamaan projektia realistisesti, elleivät testausstrategian vaatimukset ole hänellä selkeitä. Pelkästään teknisten testausvaiheiden suunnittelu ei riitä, mikäli tarkoitus on saada testausprosessi kattavammin osaksi varsinaista kehitysprosessia, kuten tässä työssä oli tarkoitus. Strategiaa tehdessä tulisi myös aikatauluttaa testausstrategian kehittämistapaamiset.

6.3 Jatkokehitysajatuksia yritykselle

Tässä kohdassa on edellisten lukujen mukaisesti käyty läpi jokainen eri testausosa-alue ja annettu jatkokehitysideoita toteutuneen testauksen pohjalta. Loppuun on lisätty aliluku, jossa ehdotetaan myös muiden kuin testaamiseen liittyvien prosessien kehittämistä.

6.3.1 Tikettitestaus

Tikettitestaus toimi testausstrategian mukaisesti. Tikettitestaus koettiin toimivaksi, mutta sitä käyttäessä huomattiin ongelmia puuttuvan dokumentaation takia. Testaajan piti testauksen aikana kysyä kehittäjiltä paljon tarkennuksia, koska tiketteihin ei ollut tarkennettu tarpeeksi, missä niitä voisi testata ja mitä kuvauksella oli tarkoitettu. Tikettitestauksessa oli vaarana, että kehittäjät eivät testaa tarpeeksi tikettejään, kun ne menivät toiselle testauskierrokselle entisen yhden kierroksen sijaan. Tällaista ei kuitenkaan tuntunut olevan havaittavissa, vaan testaajan löytämät ongelmat olivat enimmäkseen erityistapauksia.

Tikettitestaus kannattaa ottaa yrityksessä käyttöön, koska sen avulla on huomattu joitakin käytettävyysongelmia ja havaittu vikoja. Tikettitestaus sopii yrityksen prosesseihin, ja sitä pystytään tarpeen tullessa teettämään myös muulla henkilöstöllä, jos testaukseen tarvitaan enemmän resursseja. Uusi työntekijä voidaan perehdyttää nopeasti tähän testaustapaan, etenkin jos järjestelmän dokumentaatioon panostetaan.

6.3.2 Hyväksymistestaus

Hyväksymistestauksessa iso osa pitäisi saada katettua automaatiotesteillä ja päästä lisäämään tutkivan testauksen osuutta, ja tähän yrityksellä on jo selkeä suunta. Hyväksymistestaukselle tulisi olla tarpeeksi aikaa, ja versiojulkaisujen aikatauluihin tulisi kiinnittää selvästi enemmän huomiota yrityksessä. Hyväksymistestausvaiheeseen tulisi saada selkeä toimintamalli, jota noudatettaisiin jokaisessa versiojulkaisussa. Tällä tavalla pystyttäisiin todentamaan, että testausta on tehty riittävästi. Lisäksi hyväksymistestausvaiheeseen pitäisi tehdä selkeät ehdot, milloin se katsotaan riittävän kattavaksi ja versio julkaisukelpoiseksi.

6.3.3 Testiautomaatio

Testiautomaation tarvetta korostetaan nykyään paljon, ja se voikin olla ainoa ratkaisu saada katettua testeillä isot web-sovellukset, jotka ovat monimutkaisia kokonaisuuksia. Automaatiotesteissä on kuitenkin ongelmana testien ylläpidettävyys järjestelmän muuttuessa. Kohdejärjestelmä on nimenomaan koko ajan kehittyvä järjestelmä.

Automaatiotestien osalta testausstrategia epäonnistui. Tästä ei kuitenkaan voida syyttää suoraan työkalua tai sen virheellistä valintaa. Valittua työkalua ja sen rajoitteita ja sopi-

vuutta tai sopimattomuutta ei voida arvioida, koska sitä ei päästy käyttämään. On mahdollista, ettei mikään muu testiautomaatiotyökalu olisi ollut sellainen, joka näillä resursseilla ja käyttöliittymämuutoksen aikana oltaisi saatu toimimaan paremmin.

Automaatiotestien päivittäminen ja uusien luominen on avainasemassa, kun uusi käyttöliittymä on saatu julkaistua, eli elokuun versiopäivityksen yhteydessä. Automaatiotesteillä saadaan korvattua paljon manuaalista testausta, ja siten tutkivan testauksen osuutta pystytään lisäämään riittävästi. Automaatiotestejä pitäisi päästä testaamaan myös järjestelmän testiversioon, johon olisi kylvetty virheitä, joita automaatiotestien pitäisi löytää. Tällä tavalla voitaisiin varmentua siitä, että automaatiotestit toimivat odotetusti ja löytävät virheitä järjestelmästä.

6.3.4 Tietoturvatestaus

Ulkopuolisen tietoturva-arvioinnin takia järjestelmän tietoturva on hyvällä tasolla maaliskuun versiojulkaisun jälkeen. Tietoturvaan tulee kiinnittää kehitysvaiheessa huomiota, ja tietoturvaskannausta Acunetix-tietoturvaskannerilla tulisi jatkaa. Skannausympäristön version päivitys tapahtuu kätevästi samalta sivuilta kuin Beta-ympäristön päivitys, mikä on hyvä asia. Versiosta, johon tietoturvaskannaus tehdään, tulisi tehdä selkeä päätös, jotta skannausta ei vahingossa tehdä vanhaan versioon. Päätöksen pohjalta voidaan määrittää selkeät ohjeet ja käytännöt ympäristön päivittämiseen. Tietoturvaskannaus pitäisi myös tuoda selkeästi osaksi hyväksymistestausta.

6.3.5 Suorituskykytestaus

Suorituskykytestauksen puuttuminen on iso puute. Sitä ei määritelty strategiassa tarkasti, koska testiautomaatiotyökalussa on teoriassa mahdollisuus suorituskykytestaukseen, mutta sitä ei pystytty kokeilemaan puutteellisilla automaatiotesteillä.

Järjestelmän suorituskyky pitäisi arvioida yrityksen sisällä ja määrittää, onko se halutulla tasolla. Suorituskykytestaus tulisi aloittaa yrityksessä, kun testiautomaatio on saatu kuntoon ja muu testausprosessi vakiinnuttua. Suorituskykytestauksella vähintään voitaisiin pitää siitä huoli, että järjestelmän suorituskyky ei huononisi.

6.3.6 Käytettävyystestaus

Järjestelmän käytettävyyteen kiinnitettiin erityistä huomioita maaliskuun versiojulkaisussa. Käytettävyyteen tulisi myös jatkossa kiinnittää huomiota.

Käytettävyystestauksessa voitaisiin hyödyntää projektihenkilön tai myyjän osaamista. Kehitystiimin ulkopuolinen työntekijä voisi tehdä tutkivaa testausta pari viikkoa ennen julkaisua ja julkaisemisvaiheessa Gpdemo-ympäristössä. Testaus voisi olla pituudeltaan esimerkiksi kaksi 2 – 3 tunnin sessiota. Tämä ei olisi liian suuri määrä ajatellen resursseja.

Se kuitenkin lisäisi tutkivan testauksen laatua huomattavasti. Tällä tavalla saataisiin myös tuoreempaa näkemystä järjestelmän kehittämiseen ja ideoita sen parantamiseen.

Ongelmaksi voi tulla kehitystiimin ulkopuolisten työntekijöiden asennoituminen, jos heitä pyydetään tekemään heille varsinaisesti kuulumattomia tehtäviä. Putkinäkö saattaa vaivata testaajia ja kehittäjiä, kun he ovat työskennelleet version parissa muutaman kuukauden, eivätkä he osaa tarkastella sitä tuorein silmin. Myyjillä on hyvää käytännön kokemusta, miten asiakkaat käyttävät järjestelmää ja se toisi asiakaslähtöisemmän näkökulman testaamiseen.

6.3.7 Dokumentaatio

Selkein jatkokehitysidea yrityksen kehitysprosessiin on tikettien ja dokumentaation parantaminen. Dokumentaation ylläpidettävyys on tietysti ongelma, mutta jos siitä tehdään riittävän yksinkertainen, joka kattaa perusasiat sovelluksesta, ei sen päivittäminen ole liian työlästä. Paremmin tehdyt tiketit myös toimivat osaltaan dokumentaationa.

Tikettejä tehdessä pitää keskittyä käyttämään selkeää kieltä ja kuvailla ongelma niin, että myös muut ymmärtävät mikä on ongelmana. Kuvat yleensä auttavat vaikeammissa tilanteissa ymmärtämään ongelmatilannetta paremmin. Jos ja kun yrityksessä tikettien halutaan toimivan osana dokumentaatiota, tulee tikettiä luodessa kiinnittää huomiota sen hakuoptimointiin. Hakuoptimoinnilla tarkoitetaan, että tiketti löytyisi helposti siihen liittyvillä hakusanoilla, eikä esimerkiksi erityistapauksissa taivutusmuotojen takia jäisi löytymättä. Tässä on erityisen tärkeää, että tiketeissä ei käytetä sekakieltä. Yrityksessä on tapana eri henkilöillä käyttää eri termejä samoja asioita tarkoitettaessa, ja tämä vaikeuttaa tiettyyn asiaan liittyvien tikettien löytämistä.

Kehittäjien myös tulisi tikettejä tehdessään dokumentoida, jos he ovat tehneet jotain erityisratkaisuja toteutustyössä. Kehittäjien olisi myös tärkeää kirjoittaa tiketteihin, millä moduulilla tikettiä voidaan testata. Muuten testaajalta menee todella paljon aikaa siihen, että hän löytää oikean moduulin tietyillä asetuksilla järjestelmästä.

Yrityksen kannattaisi konkreettisena toimenpiteenä kouluttaa työntekijät tikettien laatimiseen. Koulutuksessa voitaisiin käsitellä esimerkkien kautta nykytilan ongelmia ja yhdessä keskustellen sopia yhtenäisestä tikettien kirjoittamistavasta. Koulutus saattaisi motivoida työntekijöitä paremmin parantamaan käytäntöjään kuin sähköpostilla tullut ohjeistus, kun he voisivat esittää kysymyksiä ja ymmärtäisivät esimerkkien kautta, miksi jotkut tiketit ovat ongelmallisia testaajille.

6.3.8 Muu testausprosessi

Testausprosessia tehostava kehityskohde on testaajien oikeuksien lisääminen. Laajemmat oikeudet antavat mahdollisuuden testaajalle rakentaa helpommin ja paremmin tarvitsemiaan testiympäristöjä. Tähän tuli jo muutosta maaliskuun versiossa julkaistun parannetun lomake-editorin kautta, joka mahdollistaa testaajan muokkaavan lomakkeiden kenttiä vapaammin kuin ennen.

Testaajat ja kehittäjät ovat helposti liian kaukana käyttäjistä. Palaverit myyjien ja kehittäjien välillä voisivat olla hyödyllisiä, kun kehittäjät kuulisivat miten asiakkaat reagoivat järjestelmään. Tämä voisi sitouttaa kehittäjiä järjestelmään paremmin. Samalla myyjät saisivat tietoa miten järjestelmää kehitetään. He pystyisivät myös paremmin ymmärtämään, miten asiakkaiden pienet pyynnöt voivat vaatia suuriakin muutoksia järjestelmään.

6.3.9 Yrityksen prosessit

Yritys voisi parantaa prosessejaan kehittämällä versiojulkaisuaikatauluaan. Selkeä parannuskohteet ovat työmääräarvioiden täsmentäminen ja versiojulkaisujen sisällön rajaaminen. Aikataulua suunniteltaessa tulee ottaa huomioon, että asiakastehtävät vaativat aikaa ja se on pois kehitystyöltä. Uusille asiakkaille pitää mahdollisesti tehdä konfigurointia, joka on myös pois muusta kehitystyöstä.

Aikataulutusongelmaa voitaisiin ratkaista muuttamalla versiojulkaisuihin liittyvää ajattelutapaa ja vakiintunutta toimintatapaa. Versiojulkaisun ajateltaisiin tapahtuvan sovitun kuun ensimmäisenä päivänä viimeisen sijaan. Tällöin tähdättäisiin siihen, että versio olisi kuun ensimmäisenä päivänä valmiina tutkivaan testaukseen Gpdemo-ympäristössä. Tällöin yllättävät aikatauluongelmat eivät haittaisi niin paljon – versio ei olisi myöhässä kun joustoaikaa olisi vielä kuukausi. Tähän kuitenkin vaadittaisiin asennemuutosta työntekijöiltä ja sitoutumista siihen, että julkaisupäivä olisi kuun ensimmäinen päivä. Muuten syntyy helposti ”onhan tässä vielä aikaa” –ajattelutapa. Yrityksen imagon kannalta olisi hyvä, että versiojulkaisujen aikataulut olisivat pitäviä eivätkä asiakkaat joutuisi odottamaan.

7. YHTEENVETO

Web-sovellusten kehittäminen ja testaus ovat nopeasti kehittyvä ala. Perinteisten sovellusten kehitystyökalut eivät vastaa web-sovellusten vaatimuksia, eivätkä vanhat testausmenetelmät pysty vastaamaan tarpeeksi hyvin web-sovellusten kehityskielten ja laatuvaatimuksien haasteisiin.

Testausstrategian luominen web-sovellukseen ei ole yksinkertainen tehtävä. Testausstrategiaa luotaessa pitää ottaa huomioon monia eri asioita. Tässä työssä kohdejärjestelmään testausstrategian luomisen erityispiirteitä olivat yrityksen pienet resurssit ja melko olemattomat testauskäytännöt. Kohdejärjestelmä oli pitkään kehitetty web-sovellus, johon tehdään versiopäivityksiä kolme kertaa vuodessa. Järjestelmään siis luodaan uusia ominaisuuksia jatkuvasti.

Testausstrategiaa luotaessa siitä pyrittiin tekemään tarpeeksi kevyt ja helposti toteutettava. Muutokset yrityksen prosesseihin pyrittiin pitämään sellaisina, että kehittäjien oli helppo muuttaa entisiä toimintatapojaan ja mukautua uusiin prosesseihin. Kehittäjät ottivatkin muutokset hyvillä mielin vastaan ja yrityksen työntekijät toivoivat, että testauksen kehittämistä jatkettaisiin edelleen.

Luotua testausstrategiaa ei saatu kehitettyä pitkälle kahden versiojulkaisun aikana. Tähän vaikuttivat inhimilliset ongelmat, kuten testaajan ja testausstrategian kehittäjän joutuminen leikkaukseen. Lisäksi ohjelmistoalan yleiset ongelmat aikataulujen venymisestä ja kiireestä vaikuttivat siihen, ettei tarvittavia muutoksia pysytty toteuttamaan.

Tärkeä osa testausstrategiaa oli automaatiotestien luominen ja käyttöönotto. Kevyt ja helppokäyttöinen testityökalu saatiin valittua, mutta sitä ei päästy todellisuudessa testaamaan tai käyttämään. Tähän vaikutti se, että sovellukseen tehtiin iso käyttöliittymäuudistus. Automaatiotestit pysyttiin päivittämään vasta muutoksen jälkeen, ja tällöin versio oli jo pitänyt julkaista, koska oltiin myöhässä aikataulusta.

Yrityksen kehitystiimi koki, että vaikka testausstrategiaa ei saatu kehittyä niin paljon kuin oli tarkoitus, testauskäytännöt ovat selvästi parantuneet tikettitestauksen avulla. Kehitystiimi myös koki, että testaukseen panostaminen on tarpeellista myös jatkossa. Lisäksi testausstrategian kehityksen aikana on huomattu muita kehitysprosessin ongelmakohtia ja niihin pystytään jatkossa puuttumaan.

Työssä esitettiin myös jatkokehitysideoita yritykselle, joita hyödyntämällä yritys pystyy arvioimaan prosessejaan tarkemmin. Arvioimalla prosessejaan ja hyödyntämällä työn tuloksia yritys voi kehittää käytäntöjään eteenpäin ja parantaa tuotteen laatua ja tehostaa kehitysprosessejaan. Tällä olisi varmasti saavutettavissa pitemmän aikavälin säästöjä ja

imagon parannusta, kun asiakkaille toimitettavat julkaisut olisivat aikataulussa ja niissä olisi vähemmän virheitä.

Tärkeä löydös työssä oli se, että testausstrategian luomisessa ei riitä pelkästään sen tekninen suunnittelu. Tässä työssä keskityttiin testausstrategian luomisessa liikaa sen tekniseen puoleen. Testausstrategiaa luodessa ja käyttöönottaessa tulee huomioida kaikki suunniteltuun testausprosessiin liittyvät henkilöt, ja sitouttaa heidät projektiin. Testausstrategiaa tulisi iteratiivisesti integroida yrityksen prosesseihin, ja ihmiset pitäisi saada työskentelemään yhdessä testausstrategian parantamiseksi sen sijaan, että testausstrategia jätetään testaajan vastuulle ja testaus eriytetään muusta kehitystyöstä. Myös testaajan tulee selkeästi kommunikoida muille, miten testausstrategiaa tulisi kehittää ja suunnitella testauspalavereita ja kehitystapaamisia osaksi testausstrategiaa.

Testauskäytäntöjen parantaminen olisi varmasti monissa yrityksissä tärkeää. Testausstrategian luomiseen tai päivittämiseen pitäisi varata resursseja riittävästi. Yrityksen tarpeiden tunnistaminen ja entisten käytäntöjen arvioiminen ovat hyvä lähtökohta testausstrategian kehittämisen aloittamiselle.

LÄHTEET

Acunetix, 2015. SQL Injection: What is it? [Online] Saatavilla: http://www.acunetix.com/websitesecurity/sql-injection/ [Viitattu 17 May 2015].

Artzi, S., Kiezun, A., Dolby, J., Tip, F., Dig, D., Paradkar, A., Ernst, M. D., 2008, July. Finding bugs in dynamic web applications. In *Proceedings of the 2008 international symposium on Software testing and analysis* (pp. 261-272). ACM.

Benedikt, M., Freire, J., & Godefroid, P., 2002. VeriWeb: Automatically testing dynamic web sites. In *In Proceedings of 11th International World Wide Web Conference (WWW'200*

Brooks, D., 2011. Guide to HTML, JavaScript and PHP. Spinger.

CGISecurity, 2002. The Cross-Site Scripting (XSS) FAQ. [Online] Saatavilla: http://www.cgisecurity.com/xss-faq.html [Viitattu 17 Toukokuu 2015].

Confluence - Team Collaboration Software | Atlassian. 2015. Confluence - Team Collaboration Software | Atlassian. [Online] Saatavilla: https://www.atlassian.com/software/confluence [Viitattu 17 May 2015].

Di Lucca, G. A., Fasolino, A. R., Faralli, F., De Carlini, U., 2002. Testing web applications. In *Software Maintenance, 2002. Proceedings. International Conference on* (p. 310-319). IEEE.

Di Lucca, G., Fasolino, A., 2006, Web Engineering chapter 7 Web Application Testing Springer Berlin Heidelberg, p. 219-260

Dobolyi, K., 2010. *An Exploration of User-Visible Errors in Web-based Applications to Improve Web-based Applications* (Doctoral dissertation, University of Virginia).

Dooley, J., 2011. Software development and professional practice. Apress.

EDInteractive, 2015. [Online]. Saatavilla: http://www.edinteractive.co.uk/article/?id=4[Viitattu 24 huhtikuu 2015].

Emery, D. H., 2009. Writing Maintainable Automated Acceptance Tests. In *Agile Testing Workshop, Agile Development Practices.*

Grannel, C., 2007. The Essential Guide to CSS and HTML Web Desing. USA: FriendsofED.

Itkonen, J., Mantyla, M. V., & Lassenius, C., 2013. The role of the tester's knowledge in exploratory software testing. *Software Engineering, IEEE Transactions on, 39*(5), p. 707-724.

Itkonen, J., Mantyla, M. V., & Lassenius, C., 2007. Defect detection efficiency: Test case based vs. exploratory testing. In *Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on* p. 61-70. IEEE.

ISTQB, 2007. ISTQB:n testaussanasto. [Online] Saatavilla: http://www.fistb.fi/sites/fistb.ttlry.mearra.com/files/istqb_sanasto.pdf

Jazayeri, M., 2007. Some trends in web application development. In*Future of Software Engineering, 2007. FOSE'07* p. 199-213. IEEE.

JIRA 2015. JIRA - Issue & Project Tracking Software | Atlassian. [Online] Saatavilla: https://www.atlassian.com/software/jira. [Viitattu 17 Toukokuu 2015].

Katara, M., Vuori, M., Jääskeläinen A., 2013. Ohjelmistojen Testaus. Tampereen Teknillinen Yliopisto, Tietotekniikan laitos. Oppimateriaali.

Kessler, D. 2012. Automated Testing Startegy for Legacy Systems. [Online] Saatavilla: http://blogs.sourceallies.com/2012/04/automated-testing-strategies-for-legacy-systems/ [Viitattu 17 Toukokuu 2015].

Khan, M. E., Khan, F. 2012. A Comparative Study of White Box, Black Box and Grey Box Testing Techniques. *International Journal of Advanced Computer Science and Applications (IJACSA), 3*(6).

Koskimies, K., Mikkonen, T. 2005 Ohjelmistoarkkitehtuurit. Talentum Media Oy.

Langer, A. M., 2012. *Guide to Software Development: Designing and Managing the Life Cycle*. Springer Science & Business Media.

Leff, A., Rayfield, J.T., 2001. "Web-application development using the Model/View/Controller design pattern," *Enterprise Distributed Object Computing Conference, 2001. EDOC '01. Proceedings. Fifth IEEE International* , p.118-127.

Lei, K., Ma, Y., Tan, Z., 2014. Performance Comparison and Evaluation of Web Development Technologies in PHP, Python, and Node.js. In *Computational Science and Engineering (CSE), 2014 IEEE 17th International Conference on* p. 661-668. IEEE.

Lengstorf, J., 2010. Pro PHP and jQuery. Apress.

Mikkonen, T., Taivalsaari, A., 2007. Web Applications: Spaghetti code for the 21st century.

Mikkonen, T., Taivalsaari, A., 2010. A. The Mashware Challenge: Bridging the Gap Between Web Development and Software Engineering. In Proceedings of the 2010 Workshop on Future of Software Engineering Research, 245-249, ACM Press

Offutt, J., 2002. Quality attributes of web software applications. *IEEE software*, *19*(2), p. 25-32.

O'reilly, T., 2007. What is Web 2.0: Design patterns and business models for the next generation of software. *Communications & strategies*, (1), 17.

Osherove, R., 2009. The Art of Unit Testing: with Examples in .NET. 1 Edition. Greenwich, CT: Manning Publications Co..

Pan, I., Mukherjee, S., 2012 Automated Test Approach for Web Based Software.

Prakash, V., Gopalakrishnan, S., 2011, April. Testing efficiency exploited: Scripted versus exploratory testing. In *Electronics Computer Technology (ICECT), 2011 3rd International Conference on* Vol. 3, p. 168-172. IEEE

Redouane, A., 2002. Guidelines for Improving the Development of Web-Based Applications. In *Proceedings of the Fourth International Workshop on Web Site Evolution (WSE'02)* (p. 93). IEEE Computer Society.

Ricca, F., Tonella, P., 2001. Analysis and testing of web applications. In*Proceedings of the 23rd international conference on Software engineering* p. 25-34. IEEE Computer Society.

SahiPro, 2015. FAQ. [Online] Saatavilla: http://sahipro.com/faq/ [Viitattu 17 Toukokuuta 2015].

Rice, R., 2012. Regerssion Testing in Large, Complex and Undocumented Legacy Systems. [Online] Saatavilla: http://www.technologytransfer.eu/article/101/2012/6/Regression_Testing_in_Large,_Complex_and_Undocumented_Legacy_Systems.html [Viitattu 17 Toukokuuta 2015]

Sacks, M., 2012 . Pro Website Development and Operations: Streamlining DevOps for large-scale websites. Apress.

Scotland, K., 2010. Aspects of Kanban. Methods & Tools - Summer 2010.

Skype, 2015. Skype/Microsoft. [Online] Saatavilla: http://www.skype.com [Viitattu 17 Toukokuu 2015].

Software Testing Help, 2015. Web Testing: Complete guide on testing web applications. [Online] Saatavilla: http://www.softwaretestinghelp.com/web-application-testing/ [Viitattu 17 Toukokuu 2015].

Software Testing Help, 2015/2. What is Software Compatibility testing? [Online] Saatavilla: http://www.softwaretestinghelp.com/software-compatibility-testing/ [Viitattu 17 Toukokuu 2015].

Storey, D., 2015. The Client-Server Model. [Online] Saatavilla: http://demosthenes.info/blog/137/The-ClientServer-Model [Viitattu 17 Toukokuu 2015].

Taivalsaari, A., Mikkonen, T., Ingalls, D., Palacz, K., 2008. Web browser as an application platform: The Lively Kernel experience.

Tian, J. 2005. Testing Techniques: Adaptation, Specialization, and Integration. *Software Quality Engineering: Testing, Quality Assurance, and Quantifiable Improvement*, p.203-219.

Vrieze M., 2012. The Difference For Test Automation Between Cutting Edge And Legacy Software. [Online] Saatavilla: http://martijndevrieze.net/2012/06/25/the-difference-between-cutting-edge-and-legacy-software-test-automation/ [Viitattu 17 Toukokuu 2015].

UsabilityNet, 2015. Heuristic evaluation. [Online] http://www.usabilitynet.org/tools/expertheuristic.htm [Viitattu 17 Toukokuu 2015]

W3C, 2008. Web Content Accessibility Guidelines. [Online] Saatavilla: http://www.w3.org/TR/WCAG20/ [Viitattu 17 Toukokuu 2015].

W3Schools, 2015. AJAX Introduction. [Online] Saatavilla: http://www.w3schools.com/Ajax/ajax_intro.asp [Viitattu 17 Toukokuu 2015]

W3Techs, 2015. Usage statistics and market share of Java for websites. [Online] Saatavilla: http://w3techs.com/technologies/details/pl-java/all/all [Viitattu 17 Toukokuu 2015]

Wilde, E., 2007. Declarative Web 2.0. *Information Reuse and Integration, 2007. IRI 2007. IEEE International Conference on* p. 612-617. IEEE.

Winkler, I. S., Dealy, B., 1995. Information Security Technology? Don't Rely on It. A Case Study in Social Engineering. In *USENIX Security*.

Xu, L., Xu, B., 2004. A framework for web applications testing. *Cyberworlds, 2004 International Conference on* p. 300-305. IEEE.

Zemin, Z., Fei, X., Fen, Z., 2011. Research on the Design of Test Strategy for WebGame. *Intelligence Science and Information Engineering (ISIE), 2011 International Conference on* (p. 265-268). IEEE.

Zhu, B., Miao, H., Cai, L., 2009. Testing a web application involving web browser interaction. In Software Engineering, Artificial Intelligences, Networking and Parallel/Distributed Computing, 2009. SNPD'09. 10th ACIS International Conference on (p. 589-594). IEEE.